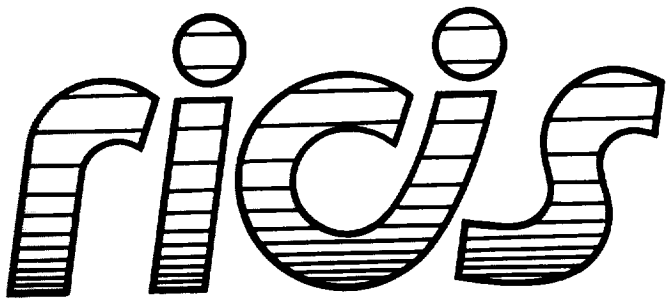
Summary Report for the Engineering Script Language (ESL)

SofTech, Inc.

November 28, 1990

Cooperative Agreement NCC 9-16
Research Activity No. SE.33

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

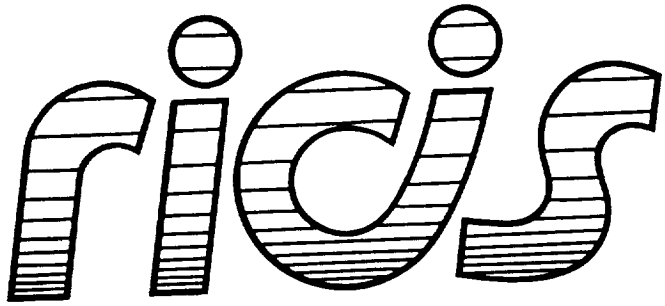
Summary Report for the Engineering Script Language (ESL)

SofTech, Inc.

November 28, 1990

Cooperative Agreement NCC 9-16
Research Activity No. SE.33

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by SofTech, Incorporated. Dr. Charles McKay served as RICIS research coordinator.

Funding has been provided by Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Ernest M. Fridge, of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

SUMMARY REPORT
FOR THE
ENGINEERING SCRIPT LANGUAGE (ESL)

28 November 1990

A UHCL/RICIS Report,
Contract No. NCC-9-16, SE.33

Prepared for:

Software Technology Branch,
Information Technology Division,
Information Systems Directorate,
Johnson Space Center

Prepared by:

SofTech, Inc.
1300 Hercules Drive, Suite 105
Houston, TX 77058-2747
(713) 480-1994

1.0 Introduction

In the past sophisticated simulation software has been used to support the generation of mission profiles for Space Shuttle planning. The simulation software Space Vehicle Dynamics Simulation (SVDS) and Flight Design System (FDS) made use of application processors and custom executives to compose complex simulations. These executives use tabular input and script files to control the execution of the simulations. The application processors rely on mission dependent and independent tabular inputs to control simulation processing. The use of these simulations as tailored by their associated tabular inputs has proven effective as a means of mission support.

Analysis of this development process has disclosed three different classes of developers and users: the software developer, the application engineer, and the operation specialist or end user. Ideally, the software developer produces/modifies the software in accordance with the specifications produced by the application engineer. The application engineer develops the software requirements based on the mission requirements obtained from the operation specialists and composes complex flight planning applications from the produced/modified software. The operation specialist uses these applications to generate and verify mission specific data.

While this process has proven effective in the past, there are limitations that may be significantly reduced with recent advances in software engineering and workstation technology. Briefly, the major limitations are:

- The textually oriented user interface of FDS and SVDS gives little insight into how to compose and implement new simulation applications.
- Application engineers must perform a large portion of the software development and maintenance because only the application engineer has the engineering knowledge necessary to understand the aerospace domain aspects of the application.
- The structure of FDS and SVDS application processors is very complex,the internal data coupling factor is high, and the large granularity of the application processors inhibits understanding of the processors and makes reuse of the application processors or their subcomponents, very difficult.

Given that new software for Space Station Freedom must be developed in Ada and that new applications may require over a million lines of code, engineers and analysts need improved tools to support their planning and analysis activities. These tools should allow the user to concentrate on the flight design and analysis activities rather than on the software engineering process involved in developing complex software applications.

To reduce these limitations, SofTech proposed to separate the concerns of the software developer from those of the application engineer by the use of an Engineering Scripting Language (ESL). The purpose of an ESL is to allow the application engineer to limit his view of the problem space to those functional concepts that are inherent in the problem space and not in the software design. Conversely, the software developer will be relieved of the task of modifying complex flight planning application and able to focus his development and modification activities to individual, nonintegrated, functionally reusable components.

SofTech's approach to the specification of an ESL technology builds upon successful past experience. SofTech has developed a proprietary graphical methodology for producing complex applications from reusable components that does not require the engineer to be knowledgeable in the underlying programming language. The Graphical Analysis and Design Technique (GADT) methodology provided a close match to the requirements of the ESL and served as an excellent starting point for the ESL specification.

The concepts underlying GADT come from both the Anti-Submarine-Warfare (ASW) Common Operational Software Support System (ACOS) and the Structured Analysis and Design Technique (SADT). ACOS was jointly developed by SofTech and the Naval Research Laboratory for U.S. Navy signal processing computers. It was developed to allow signal processing engineers, rather than programmers, to develop real-time, complex applications more quickly and at lower cost. The Navy reports that the operational use of this concept has dramatically increased the level of code reuse and the productivity of signal application designers and programmers. ACOS is the primary method the Navy plans to use for developing future signal processing applications.

SofTech extended the concepts of ACOS in its GADT Internal Research and Development (IRAD) effort to provide a unified methodology for analysis, design, and implementation. The experience gained on the GADT IRAD proved directly applicable to the analysis and definition of the ESL.

Additional requirements for the ESL have resulted from the specific nature of the flight planning applications and the anticipated environment for ESL development and use as specified by the Software Technology Branch, NASA/Johnson Space Center (JSC). The specification of an ESL conceptual model to meet the stated mission requirements is contained in this report.

2.0 ESL Methodology Concepts

The main goal of the ESL methodology is to enable an application engineer to develop flight planning applications with reusable units corresponding to functional concepts natural to his problem domain. Since the application engineer's domain is astronautical, not software, the methodology must submerge the software aspects of the problem. The methods used by the ESL to accomplish this are discussed in the following subsections.

2.1 Graphs

The ESL and its related components are meant to serve as an application generator for flight planning applications. The ESL itself is a graphical language designed to represent the applications. An ESL application represented in this graphical form is called a **graph**. A graph represents the structure (or sub-structure) of an application and consists of a set of **nodes** representing the processing elements of the graph and a set of **queues** representing the directed flow of information through the graph. A graph may be represented as a single node which is the parent node of the graph. A graph which is the underlying representation of a node in another graph, is called a **subgraph**. Subgraphs allow hierarchical structures of graphs and constitute reusable, application building blocks. Only noncyclic subgraph networks may be specified. At the lowest levels, the nodes of a subgraph denote **primitives**, processing elements composed of compiled code as opposed to further subgraph decomposition.

A hierarchy of graphs is represented at the highest level as a single node, as illustrated in Figure 2.1.-1. A single parent node, with an underlying hierarchy of subgraphs, is referred to as a graph system.

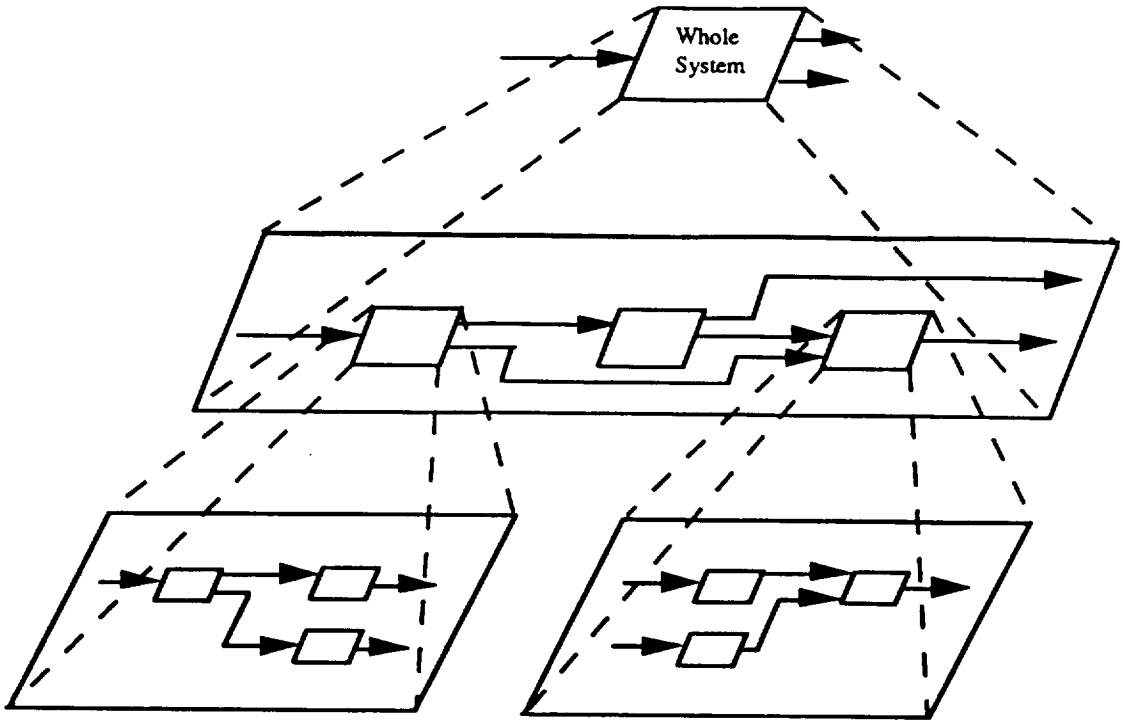


Figure 2.1-1 A Graph Hierarchy

Graph systems may be further connected with queues to form more complex systems. These complex systems are also represented as a single node.

A subgraph may accept inputs and outputs via **graph ports**. These ports logically link the ports of the parent node with input ports of nodes within the subgraph (representing the node's underlying representation) and facilitate the hierarchical nesting of subgraphs within nodes. A node port may be linked to a graph port to support hierarchical linkage between the ports of a graph's parent node and the ports of nodes within the graph.

The use of graphs allow complex applications to be described in a manner that the application engineer finds very natural to deal with. For instance, let us examine a simplified graph system for a flight planning simulation depicted in Figure 2.1-2. This graph clearly shows the main functional components of this application as well as the relative sequencing between them. The *Initialization* node contains the mechanisms to get the input data from whatever source has been intended. *The Read Phase Input* node gets the input for that phase of the flight trajectory to be simulated. The *Phase Simulation* node contains the actual flight simulation mechanisms. Finally, the *Iterator* node causes control to loop back to allow *Read Phase Input* to read the data for the next phase.

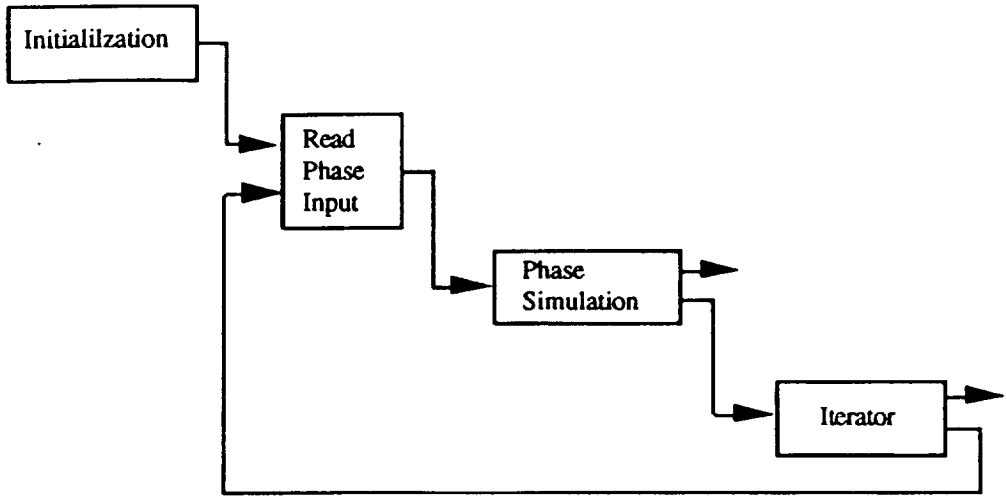


Figure 2.1-2 Top Level of a Flight Planning Application

Besides being very intuitive, the graph also represents a reusable structure for a class of similar applications. It also localizes the areas where change may be affected. An application engineer wanting to use a different atmospheric model could use a similar application to serve as the starting point for the modification process. The engineer would immediately realize that his desired change would not impact the *Initialization*, *Read Phase Input*, or *Iterator* nodes so he would examine the decomposition of the *Phase Simulation* node depicted in Figure 2.1-3.

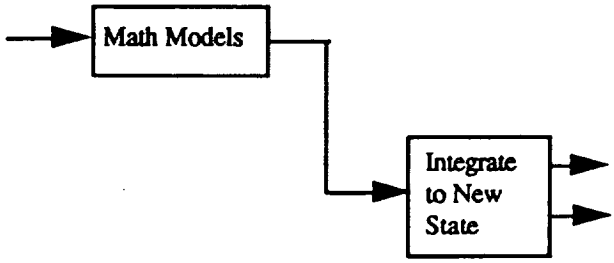


Figure 2.1-3 Phase Simulation Subgraph

Looking at this subgraph the application engineer is again immediately cognizant of the application structure at this level and that his desired change is localized to the *Math Models* node. If the desired change had been to use a different kind of integration then the change would be made to the *Integrate to New State* node. For our example, however, we would decompose the *Math Models* node. This decomposition is shown in Figure 2.1-4.

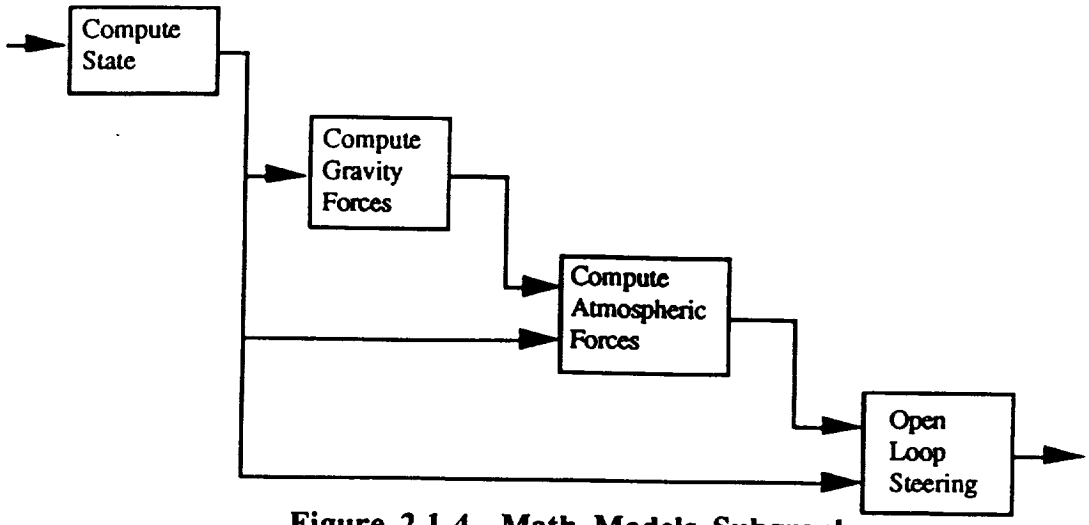


Figure 2.1-4 Math Models Subgraph

In the *Compute Atmospheric Forces* node we'll assume for our example that we have found the proper place to substitute the new atmospheric model. It is immaterial whether the new atmospheric model is a subgraph or a primitive. The engineer would use the ESL Editor (discussed below) to detach the present underlying representation for *Compute Atmospheric Forces* and then attach the new atmospheric model.

Again, the structure of the application at the level shown in Figure 2.1-4 is very understandable. Even an application engineer with no prior experience with this application can gain some familiarity merely by inspection. Once an application of this class is understood then all applications in the class become accessible to the engineer. This "quick startup" factor reduces the time for application training - another intended feature of the ESL.

In addition, a software engineer responsible with maintaining primitives in this application doesn't need to know anything about the application or its structure. Providing the primitive with the computational capability to supply the proper data on its ports, is the software engineers concern. In this sense, the ports of a primitive or subgraph constitute it's only interface to other application components. This lessens the chances that unanticipated "side effects" from a software change could affect an application.

2.2 Data Flow Principles

The ESL shall allow software applications to be described as a hierarchy of data flow graphs in a manner similar to block diagrams. A data flow graph consists of a number of nodes connected by a set of directed edges referred to as queues since they carry data on a first in, first out (FIFO) basis. A node represents an operation while a queue represents the directed flow of data between nodes. These concepts are illustrated in Figure 2.2-1.

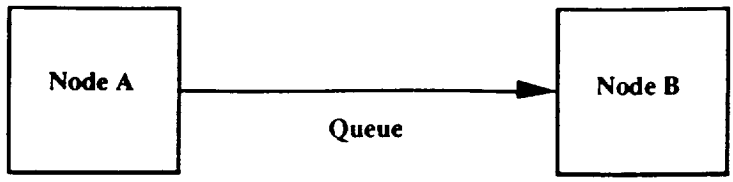


Figure 2.2-1 Basic Data Flow Between Nodes

From a user's perspective, one does not need to be concerned about the inner structure of a node. Its execution behavior may be specified at the node level, and it may be reused across different applications and treated as a black box.

The computational model of a data flow graph is based on data flow rules. These rules specify that a node with an underlying operation is ready for execution when:

- All of its incoming queues contain the required amount of data for the processing of the underlying operation.
- All of its output queues may accept data.

The ESL Shell (discussed in Section 3.2) manages the execution of the nodes.

The activation of a node does not require its underlying operation to be referenced by another operation. Rather, the sequencing of the node execution is based on data flow rules and the data flows between nodes as specified by the graphs.

Each node has a number of input and output ports from which data may flow into or out of the node. Each queue has a head and a tail. The head represents the end of the queue from which data flows out or is dequeued. The tail represents the end of the queue which data flows into or is enqueued. Each queue head may be attached to an input port of a node, while each queue tail may be attached to an output port of a node. These concepts are illustrated in Figure 2.2-2.

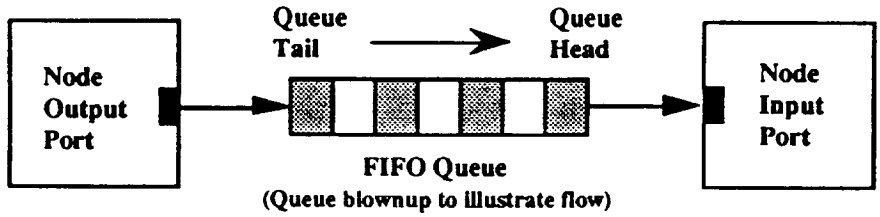


Figure 2.2-2 Moving Data Through a Queue

A node has a set of **Node Execution Parameters** (NEPs). The priority level of the node is one such parameter. Additionally, there is a set of NEPs for each port of the node.

For input ports, these parameters refer to:

- The minimum number or threshold of data elements which must be queued up on the input queue before the node is ready for execution.

- The number of data elements which may be read one or more times from the node's input queues and processed by the node's underlying operation.
- The number of data elements which may be skipped before starting the read.
- The number of data elements which may be consumed and thus removed from each of the node's input queues.

A data element is the basic unit of information in a queue, equivalent to one item of the declared data type.

For output ports the NEPs refer to:

- The number of data elements which may be produced one or more times onto each of the node's output queues.

The production of output information provides input information to a node downstream in the graph and may make it ready for execution by meeting or exceeding its threshold NEP. The production may also block the executing node if the output queue cannot accept more data. The consumption of input information reduces the total amount of information resident on an input queue which may cause it to drop below the threshold associated with the node's input port. The consumption may also make an upstream node ready for execution if it had been blocked by an output queue not able to accept data.

During the execution of a node the values of the NEPs are used by the node's underlying operation to read, consume, and produce information. The mechanism for an operation to effect a read, consumption, or production of information is by means of ESL command statements (see Sections 3.2.4 and 5.0). These statements can be executed numerous times and in any order that is logically consistent with the node's underlying operation. Command statements are used, to evaluate the current value of the NEPs and change the values of the NEPs. If the values of the NEPs are changed, the new value of the NEPs are used in subsequent read, consume, and produce processes for current execution or subsequent executions, while the new threshold value is used in the next node execution criteria.

2.3 Application Generation and Execution Using ESL

The generation of executable applications using ESL involves several components, these components being the:

- ESL Editor
- Library and Library manager
- Graph Translator
- ESL Shell
- High Order Language (HOL) compiler.

The ESL Editor comprises the user interface and provides the means for graphical representation and manipulation of the nodes and queues constituting an ESL application as discussed above. The functionality of the ESL Editor is specified in section 4.0.

2.3.1 Graph Schema

ESL graphs are the means by which the user defines the application. The multiple ESL graphs as well as the the associated information that must be present for each graph are stored in a component library and managed by a library manager. The graphs and their associated information are represented in the library as **graph schemas.**

The ESL system shall use the Bauhaus system to manage the ESL graphs and associated data. To do this the ESL editor must be able to access and manipulate the Bauhaus data structures, called schema. This section discusses the Bauhaus system and the conceptual representation of ESL information using Bauhaus schema. The accessing of the Bauhaus schema by the ESL editor is discussed in Section 4.0.

2.3.1.1 ART

The Bauhaus is based on Inference's Automated Reasoning Tool (ART). ART is a software tool kit for building expert systems. ART offers advanced features such as symbolic modeling using schema, facts, and rule-based reasoning with forward and backward chaining and pattern matching. "A schema is a collection of information about a particular object stored in the database."[1] A schema has a name and contains information such as object attributes and relations to other objects. Schema also allow the notion of inheritance through the definition of subclasses. Subclass schemas inherit the types of attributes and relations from their parent classes.

2.3.1.2 The Bauhaus System

Bauhaus is an application of the ART system oriented toward software and data reuse. Bauhaus assumes a software development reuse paradigm of matching components from a catalog with a selected specification and reusing a component matching the specification or modifying and reusing the closest matching component if no exact match exists.[2] To accomplish this, a knowledge representation for each catalogued component must be created using the ART schema system. This knowledge representation captures the attributes and relationships for each component as well as the constraints for its reuse. Bauhaus also features a template-based code generation capability.

2.3.1.3 Bauhaus as ESL Library Manager

The Bauhaus system shall be used as library manager for the ESL graphs and associated tables. Associated with the top level (applications) graphs, subgraphs, and primitives, are Bauhaus schemas. This concept is illustrated in Figure 2.3.1.3-1. A graph schema represents the items found on a graph by the attributes included as schema. The attribute values often reflect the name of the graph object and often reference the schemas or tables associated with that graph object. For instance, suppose that the attribute *Node_Schema* must be included for all graph schema. When a graph for a particular application, such as *Top_Level_App*, has three nodes defined, the schema for this graph has three *Node_Schema* attributes. The attribute values for the *Node_Schema* attributes are

---

[1] Extracted from the ART Programming Language Reference Manual by Sherry H. Walden

[2] "Automated Software Development Workstation Project, Phase II Report, NAS -17766" delivered by Inference Corporation

Input_Processor , *Phase_Execution*, and *Monte_Carlo*. Each of these is a pointer to the schemas for the respective subgraphs. Furthermore, if the node *Monte_Carlo* is a primitive node, then the *Monte_Carlo* schema has the attributes Source and Executable. The values of these attributes point to the source and executable files for this primitive.

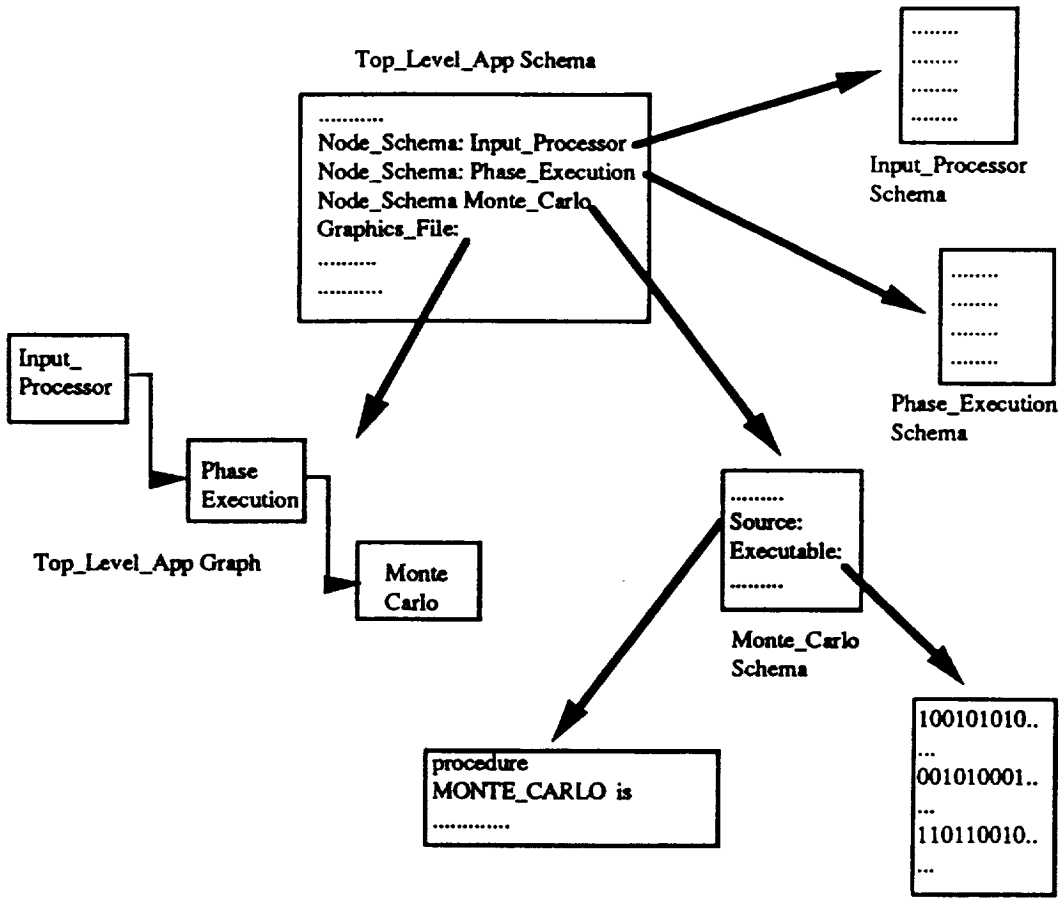


Figure 2.3.1.3-1 A Graph Representation in Bauhaus Schema

A schema also has other attributes defined that point to other graph objects such as queues. In the same fashion as above, the presence of a type of attribute denotes that: 1) the graph has that object on it; 2) the name of the attribute value is the graph name for that object; and 3) the attribute value points to an information table or other data structure associated with that object. This associated information may be stored in a data structure particular to the ESL system or, as in the above case of the procedure source code and executable, some other representation. In each case, however, a Bauhaus schema is used to reference the appropriate information.

2.3.2 Graph Implementation and Execution

Prior to executing a completed application, the graphs must be translated to a high order language (HOL) representation and subsequently compiled. A **graph implementation** is an HOL representation of a hierarchical ESL data flow graph, that can be compiled by a standard HOL compiler for subsequent execution.

The translation process generates the graph implementation by mapping the features found in the application's graph schemas to predefined HOL constructs. The graph implementation must create the run-time structures and preserve the run-time semantics of the data flow paradigm used by the ESL. In a graph implementation, ESL Shell operations are implemented by a collection of procedures called ESL command statements. A graph implementation is conceptually composed of three parts:

- ESL command statements that are used to build the data structures used by the ESL graphs.
- ESL command statements that are used by the primitives to access the ESL data structures.
- An ESL executive which shall control the execution of an ESL graph, preserving the general principles of the data flow paradigm.

Collectively these are known as the ESL Shell services. The collection of Shell operations and the HOL semantics are the interface between the data flow graph and the underlying target system. The ESL Shell is discussed in Section 3.2. That portion of the Shell services constituting the ESL command statements are specified in Section 5.0.

The translation process discussed above, involves no ESL functionality or run-time execution characteristics. The functional requirements of the translator may be completely described as the translation of the graph schemas and associated data into the graph implementation. The functional specification of the translator is composed of the graph schemas and the ESL Command Statements discussed in Section 5.0. The translation process itself is highly dependent on the features and constructs in the HOL chosen for the graph implementation and the tools that are available for use in performing the translation. The design dependent aspects of the translation process are discussed in Section 6.0.

When the graph implementation is compiled and linked, the resultant load file is called the **graph executable.** At run-time, the structures representing nodes and queues are created. These created structures are used by the Shell for real-time execution and reflect an expanded, one level data flow graph where each node has only an underlying primitive operation. After a graph executable is elaborated, the identification of its parent node may be passed to the Shell for execution which then allows data to start passing through the graph. The run-time management of graphs, node execution, and queue management are provided by the ESL Shell and the HOL run-time operating system.

3.0 ESL Specification

The ESL is comprised of two parts: the components of the ESL, and the ESL Shell. This section specifies both the ESL components and the ESL Shell.

3.1 ESL Components

The ESL components consist of the objects that must be dealt with while using the ESL Editor. The basic ESL objects consist of nodes and queues.

3.1.1 Nodes

Each node in a graph represents a specific operation referred to as the underlying operation of the node. The underlying operation may be either a graph (a subgraph) or a primitive operation. Subgraphs represent complex portions of the application and are graphs in their own right. Subgraphs were discussed in Section 2.1.

A specialization of a primitive node is the selector node. Selector nodes offer a conditional selection as a graph editing capability. Selector nodes are graphically rendered as nodes having one input port and multiple outputs ports (they may be further distinguished by a different shape such as an oval instead of a box) An example of a selector node represenation is shown in Figure 3.1.1-1. The selector node's function corresponds to an IF .. THEN capability in most languages and to a CASE statement in Ada. The capability for comparison is, however, somewhat more limited since only single data values that would be passed in through the input queue would be available for comparison. This eliminates the opportunity for complex expression evaluation as the condition for selection. For this reason selector nodes are considered as nonessential in any ESL implementation and may be replaced by primitive nodes containing the desired selection criteria that are written and compiled like any other underlying primitive representation.

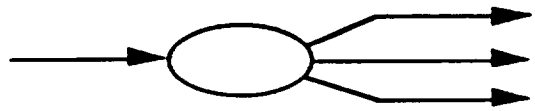


Figure 3.1.1-1 A Selector Node

3.1.1.1 Ports

Each node has a specified number of input and output ports which provide a means for the node's underlying operation to communicate with the rest of the system. Associated with each port are the data type and Node Execution Parameters, or NEPs, which specify how the data elements enqueued on an attached queue are used by the node. A data element is the basic unit of information in the queue, equivalent to one item of the declared data type.

The data elements may be moved or used by means of command statements. Command statements are the only means to transfer data between a node's underlying operation and attached queues.

A queue may be attached to a node port or the port may be unused. An unused port specifies the absence of a connection. Any command statements which attempt to read or consume data from an unused input port results in a run-time error (see Section 3.2.5). Any command statement which produces data to an unused output port results in the data being discarded.

There are four NEPs for each input port and one for each output port. The NEPs are specified for each port in which a queue is to be attached or is currently attached. NEPs which are specified for unused ports have no meaning and are ignored by the Shell for scheduling purposes until a queue is attached.

The input NEPs are:

- **Threshold** amount represents the minimal number of data elements that must be present on the corresponding input queue before the underlying operation may become ready to execute.

- **Read** amount specifies how many data elements the node's underlying operation will read from the corresponding input queue during the next execution of a command statement which reads data from the associated input port.

- **Offset** amount specifies the number of data elements on the corresponding input queue to skip before starting the read.

- **Consume** amount specifies the number of data elements to be removed from the corresponding queue on the next execution of a command statement which consumes data from the associated input port.

The output NEP is:

- **Produce** amount specifies the number of data elements which is to be produced on the corresponding output queue during the next execution of a command statement which produces data to the associated output port.

To further clarify these concepts a short example will be given. This example is based on the assignment of NEPS shown in Figure 3.1.1.1-1. The example assumes fixed data arrival rates for simplicity's sake.

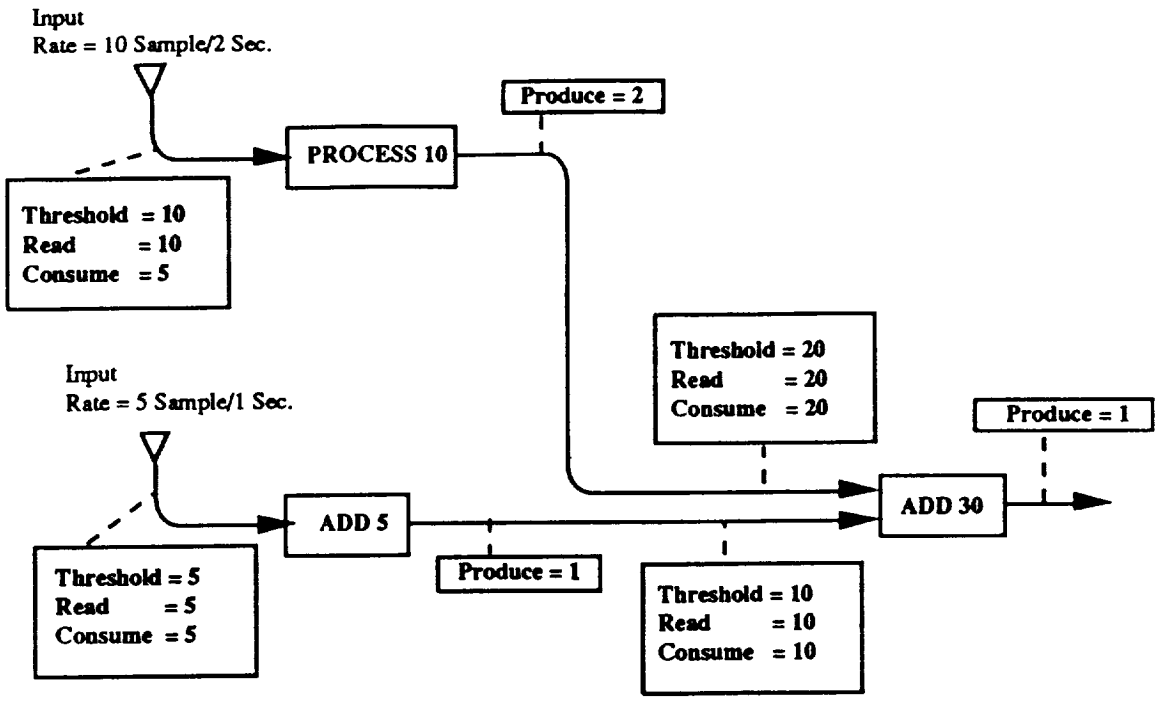


Figure 3.1.1.1-1 Moving Data as Specified by NEPs

In this example the two nodes *PROCESS 10* and *ADD 5* both make input to the node *ADD 30*. The inputs to *PROCESS 10* and *ADD 5* may arrive asynchronously, but at the rates specified in Figure 3.1.1.1-1. The node *PROCESS 10* will execute when there are 10 items in the queue (since that is *PROCESS 10*'s only queue and its threshold is 10). This will occur for the first time after 2 seconds, but thereafter every 1 second. This happens because *PROCESS 10* only consumes 5 data items each time, therefore, when 5 additional items are added to the queue (at the rate of 10 items every 2 seconds) the node is ready to fire again. Since *PROCESS 10* produces 2 data items every execution, it will output 2 data items every second after 2 seconds.

When the node *ADD 5* executes, it consumes all of the data in its only queue. This occurs every time 5 data items are accumulated which is every second according to the data arrival rate. Therefore, it outputs 1 data item every second.

The node *ADD 30* accumulates data in its queues until each queue reaches or exceeds its threshold amount. On the queue connecting *ADD 30* with *PROCESS 10* this occurs every 10 seconds since 2 data items arrive every second and the threshold amount is 20. On the queue connecting *ADD 30* with *ADD 5* this also occurs every 10 seconds. This means that *ADD 30* will produce 1 data items every 10 seconds. If the arrival rates for input data to *ADD 30* were not the same for both queues, you would eventually exceed the capacity of one of the queues if the execution persisted for long. It is the responsibility of the engineer to insure that this does not happen.

If the node has an underlying primitive operation then the Read, Consume and Produce are assigned the value of one and the Offset is assigned the default value of zero when the node is instantiated. The threshold is assigned the value equal to the maximum of (read + offset, consume). If the node has a subgraph as its underlying operation, then the NEP values are inherited from the node ports in the parent subgraph (which are linked to the corresponding

graph port during instantiation). When a NEP value is specified by means of a command statement then the value is assigned to the node port.

There is an additional NEP associated with the node which specifies the execution priority of the node's underlying operation. If the priority of a node with a subgraph as its underlying representation is specified, then all nodes in the subgraph are assigned the priority of the parent node. If nodes in the subgraph also have subgraphs as underlying representations, then the priority assignment also applies to their subgraphs. The inheritance of priority continues down the hierarchy until all nodes have primitive operations as their underlying operations. If not specified, the priority level for nodes with underlying subgraphs is undefined.

The following events characterize the execution of a node's underlying primitive operation:

- For each input port from which data is to be read from a connected queue, a read operation is effected.
- For each input port from which data is to be consumed from a connected queue, a consume operation is effected.
- For each output port to which data is to be produced to a connected queue, a produce operation is effected.
- For each NEP associated with each port, its current value may be evaluated.
- For each NEP associated with each port, its value may be assigned.
- Other ESL command statements may be invoked.
- An arbitrary set of HOL procedures or functions may be executed.

The ESL imposes no restrictions on the number of times or the order in which the above events may occur during the execution of a node's underlying operation.

3.1.1.2 Primitive Operation

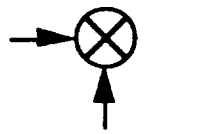
The primitive operation is the fundamental unit of processing in an ESL-designed application. The function performed by a primitive operation may be something simple, such as adding data from input queues, or something quite complex, such as a hierarchy of called procedures commonly found in structured programming. The primitive operation also includes the command statements for reading, consuming and producing data and other functions. Regardless of the function, primitive operations are the basic, reusable components in an ESL application.

Data that is read or produced by a primitive may represent information to be transformed. Alternatively, data may represent control information which affects the functionality of the primitive. Data may be references (pointers) to variables which are dynamically created by one primitive operation and passed, via queues, to another primitive operation. References to ESL graph structures are merely references to variables which are linked to represent the application data flow structure. Data may be processed in any manner consistent with the HOL. The same primitive operation may represent the underlying operation of more than one node.

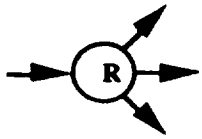
The implementation of primitive operations shall be consistent with the HOL.

3.1.2 Queues

A queue on an ESL graph represents the directed flow of information from node to node within a graph. Queues carry information on a first-in, first-out basis. Queues may be merged and replicated as indicated in Figure 3.1.2-1. A merged queue takes the input from multiple output ports for appending to the tail of the queue. The information deposited on the queue is the sum of all information sent from all connecting nodes. A replicator queue has multiple input ports (on nodes receiving information from the queue) attached to the single queue. A node belonging to any connecting port may remove information from the queue. This may affect the execution eligibility of other connecting nodes. Queues are named on an ESL graph. A name given to the queue by a user should reflect the type of data that the queue will contain. Queue names facilitate the matching of ports to queues.



Merge - Two output ports mapping to the same queue



Replicator - Replicates data to additional input ports

Figure 3.1.2-1 Merged and Replicator Queues

At run-time, each queue is a structure which is created by a command statement. The identifier of the queue is returned to allow the queue to be connected to nodes. When a queue is created, the data type of the elements to be transported over the queue and its capacity are specified. The capacity represents the maximum number of data elements which may be on the queue at any one time.

Each queue may may be defined to carry information of any data type, but the data types of its source and sink must agree. A source or sink may be a node connected to the queue or a different node which has an underlying primitive operation with a command statement having access to the queue identifier. An ordered set of data elements may be supplied by the source to a queue by the execution of a command statement. A number of elements may be removed by the sink from the head of a queue by the execution of a command statement.

A specialization of the queue is the trigger queue. A trigger queue is a typeless queue used to establish a particular execution sequencing between two or more nodes. Trigger queues appear graphically as other queues with the exception that they are not named and and the lettter T (denoting trigger) appears on the graph next to the queue construct (i.e. the arrowed line).

3.2 Shell

The Shell is a collection of procedures which support ESL graph instantiation, modification, execution, and data buffering and movement of data elements to and from queues. The precise division between the Shell software's and the operating system's functionality depends on the target machine's operating system.

3.2.1 Node Scheduling and Execution

The scheduling and execution of nodes within the graph is based upon data availability. The responsibility for maintaining node scheduling and execution rests with the Shell. Node execution will terminate either after correct execution, whenever execution is aborted, or upon completion of error handling and reporting (see Section 3.2.5).

While a graph is executing (data is flowing through the graph) the Shell performs the function of determining when a given node is ready to execute. Subsequently, the Shell must take the necessary actions to cause the execution of the node's underlying operation. While a node with a subgraph for an underlying operation behaves as a single node as far as the user is concerned, such a subgraph acts like a macro definition in that it is replaced by a more complex network of nodes. Some of these nodes may have underlying subgraphs and are, in turn, expanded like macros. The expansion is complete when a graph contains only nodes with underlying primitives. Each of these nodes will execute independently and subgraphs will no longer be distinguishable to the Shell.

Information in a graph flows from node to node via the interconnecting queues. As previously stated, a node's underlying operation is ready to execute when each of the node's input queues contains at least the threshold amount specified by the associated NEP and each of the node's output queues can accept data. An output queue may accept data if its capacity requirement has not been exceeded. The capacity requirement specifies that the queue's current size plus the produce amount of the source node is less than or equal to the queue's capacity.

Once the scheduling criteria have been met, the Shell schedules the node's underlying operation by submitting it to a prioritized node execution list. The level of priority is specified by the node's priority level NEP. These prioritized lists maintain the identification of each operation which is ready for execution. The number of priorities is implementation dependent, but the implementation shall support at least two levels of priorities. After a node's underlying operation has completed execution, the next node with the highest priority is removed from the node execution list and its underlying operation is then dispatched for execution.

A node may be on only one execution list and may be listed only once. A node, with an underlying operation which is executing, is by definition not ready and is not on any node execution list. The occurrence of one of the following events is necessary, but not sufficient, for a node to be made ready for execution:

- After a consume operation the number of remaining elements on the associated input queue causes the queue to fall below the capacity requirement. This sets the upstream node to a ready condition if all of its input data requirements are satisfied and all remaining output queues are below the capacity requirement.

- After a produce operation the number of remaining elements on the associated output queue causes the queue to go above the threshold requirement specified by the threshold NEP of the respective input port of the downstream node. This sets the downstream node to a ready condition if all remaining input data requirements are satisfied and all output queues are below the capacity requirement.

- After a node's underlying operation has executed, the node is put on an execution list if it continues to satisfy the scheduling criteria.

- After a queue is disconnected from a node, the node may then meet the scheduling criteria and be put on a node execution list.

- During graph start up the Shell may put any nodes which meet the scheduling criteria on a node execution list.

The time at which the scheduling criteria are checked for the execution node is after its underlying operation has executed. The scheduling criteria for neighboring nodes are checked during the consume and produce operations.

3.2.2 Queue Management

The Shell is solely responsible for the management of queues. The functions performed on a queue include removing data from the head of a queue and adding data to the tail of a queue. All queue management functions shall be non-interruptible with respect to other queue management functions on the same queue. The Shell shall be capable of servicing requests from the command statements to read information from the head of the queue, write information onto the tail of a queue, and delete elements from the head of a queue. Reads shall be performed in a nondestructive manner (i.e., read, consuming no data) with a specified offset indicating the number of data elements to skip prior to starting the read.

An error occurs whenever the queue from which data is to be read or to be placed, does not exist (see Section 3.2.5).

3.2.3 Graph Management

Graph instances are built at run-time by the graph executable, the compiled and linked form of the graph implementation. The graph implementation is a static description of the graph topology. Graph instances are built by the execution of a sequence of command statements (in the graph implementation) which instantiate nodes and queues and connect them together. The instantiation of a node requires the execution of a command statement which specifies the node's underlying operation as either a subgraph or primitive operation. If the statement specifies the underlying operation as a subgraph, then as part of the instantiation of the node its underlying subgraph is also instantiated. This is a recursive descent process which terminates when the complete, underlying hierarchy of subgraphs have been instantiated and all nodes at the lowest level of the hierarchy have underlying primitive operations. Once a node is instantiated, regardless of its underlying operation, its identification is returned to the Shell.

All graph instances are instantiated as a result of the instantiation of the graph's parent node. When a structure representing an instance of an ESL graph is built, an identification is returned via the parameters in the command statement creating the graph structure. This

identifier represents the identification of the graph's parent node. The parent node identification is used for all subsequent operations affecting topology or control that are made on nodes and queues.

The execution of a graph system may be started or stopped by passing the identification of the node which represents the graph system to the Shell. A command statement is used to effect this operation. The execution of a graph is the execution of the underlying operation of the nodes within it.

Although ESL graph execution is driven by data availability to nodes, the Shell does provide means for command statements to monitor and control the execution of the application system.

3.2.4 Command Statements

Command statements represent Shell services for building and controlling graphs. It is important to realize that the execution of command statements occurs at run-time. The sequential arrangement of command statements present in the graph implementation represents the instructions to build the run-time structures, connect them, pass data through the graphs, and control the graphs. Some command statements are executed before any of the primitives and are used to build and connect the run-time structures. Other command statements are used as part of a node's underlying primitive operation and hence execute as part of a primitive operation. Some of these statements move data through the graph by reading and depositing data on the queues; others may cause the halting of execution due to an error condition.

There are four main classes of command statements. The following summarizes the minimal set of capabilities for each class of command statement.

1) Statements to create the topology of the graph system:

   - Instantiate a node with a subgraph or primitive operation .
   - Create and connect queues.

2) Provide support to move data to and from queues:

   - Read with offset, consume, and produce data onto a node's input and output queues without reference to the node's underlying primitive operation or the NEP values.
   - Read with offset, consume, and produce data onto a queue which is not attached to the node.
   - Flush a queue to remove all or part of the data from the queue.
   - Initialize a queue with data. This is a combination of flushing a queue and placing data onto a queue.
   - Move data from an input queue directly to an output queue.

3) Evaluate/assign values to Node Execution Parameters:

- Evaluate current NEPs for the priority level and the threshold, read, offset, consume, and produce values for each port.
- Assign new NEP values.
- Increment, reset and evaluate a counter which is available to track the number of times a node has executed.

4) Start or stop executing particular graph systems and provide state information about a node or queue:

- Start or stop the execution of an instantiated graph system. The graph system may be restarted without loss of data within the graph system.
- Determine if a node is ready to execute.

To illustrate the use of these commands statements an example is given below showing those portions of a graph implementation that build the part of a graph system featured in Figure 3.1.1.1-1. In this example we show the command statements that build the nodes featured in the graph and specify the interconnectivity between those nodes. This specification includes the NEPs on ports that connect to queues created in our example. To fully understand all commands and types used in the example one should refer to the ESL Command Statements specified in Section 5.0.

```
---------------------------------------- declarative portion ----------------------------------

   PROC_10: NODE_ID := CREATE_GRAPH( PARENT_NODE => NIL);

-- Here we assume that PROCESS 10 node has a subgraph as an underlying
-- representation. The value of NIL indicates that this is a top level graph.

   ADD_5: NODE_ID := CREATE_NODE( PARENT_NODE => NIL,
                                  OPERATION_NAME => "ADD5");

-- The node ADD 5 is a primitive node.

   ADD_30: NODE_ID := CREATE_NODE( PARENT_NODE => NIL,
                                   OPERATION_NAME => "ADD30");

-- The node ADD 30 is also a primitive node.

   PROC10_OUT1: OUTPUT_GRAPH_PORT;

   ADD5_OUT1: OUTPUT_PORT;

   ADD30_IN1: INPUT_PORT;

   ADD30_IN2: INPUT_PORT;

-- Ports would be listed on the graph schema for each node. These would be known
-- apriori since the input and output of any primitive would need explicitly specified
```

```
-- when the primitive is written.  At the time that the graph implementation is being
-- built, all primitive nodes would be mapped to existing primitives.

   package INTEGER_Q is new QUEUE_COMMANDS( QDATA => INTEGER);

   package MEAN_MODE_Q is new QUEUE_COMMANDS(
            QDATA => MEAN_MODE_TYPE);

-- MEAN_MODE_TYPE is the type of the output of node PROCESS 10 and would
-- appear in the graph schema for this node  It would be declared in another package
-- that would be "with'ed in" by this unit.

   MM_Q: QUEUE_ID := MEAN_MODE_Q.CREATE_Q( CAPACITY => 40);

   SUM5_Q: QUEUE_ID := INTEGER_Q.CREATE_Q( CAPACITY => 20);

-- These declarations create the queues passing data from PROCESS 10 and ADD 5
-- to ADD 30.  MM_Q is the queue connecting PROCESS 10 to ADD 30 and SUM5_Q
-- is the queue connecting ADD 5 to ADD 30.

---------------------------------------- executable portion ----------------------------------

-- These statements connect the structures created above and set the execution parameters
-- for the node ports.  The procedures setting the execution parameters could be executed
-- in the primitive nodes containing those ports, if desired.

   MEAN_MODE_Q.CONNECT_OUTPUT_Q( QUEUE_NODE => MM_Q,
            NODE_NAME => PROC_10,
            PORT => PROC10_OUT1);

   MEAN_MODE_Q.CONNECT_INPUT_Q( QUEUE_NODE => MM_Q,
            NODE_NAME => ADD_30,
            PORT => ADD30_IN1);

   INTEGER_Q.CONNECT_OUTPUT_Q(QUEUE_NAME => SUM5_Q,
            NODE_NAME => ADD_5,
            PORT => ADD5_OUT1);

   INTEGER_Q.CONNECT_OUTPUT_Q(QUEUE_NAME => SUM5_Q,
            NODE_NAME => ADD_30,
            PORT => ADD30_IN2);

   NEW_PRODUCE_NEP( NODE_NAME => PROC_10,
            PORT => PROC10_OUT1,
            NEW_VALUE => 2);

   NEW_PRODUCE_NEP( NODE_NAME => ADD_5,
            PORT => ADD30_OUT1,
            NEW_VALUE => 1);
```

```
SET_NEPS( NODE_NAME => ADD_30,
          PORT => ADD30_IN1,
          NEW_READIN => 20,
          NEW_OFFSET => 0,
          NEW_CONSUME => 20,
          NEW_THRESHOLD => 20);

SET_NEPS( NODE_NAME => ADD_30,
          PORT => ADD30_IN2,
          NEW_READIN => 10,
          NEW_OFFSET => 0,
          NEW_CONSUME => 10,
          NEW_THRESHOLD => 10);
```

---------- end example ----------

3.2.5 Error Handling

The ESL necessitates the detection, notification, and recovery of certain data flow errors beyond those specified by the HOL. The following identifies the minimal class of errors which are trapped and handled by the Shell:

- An attempt for a command program to read or consume data which does not exist on the associated queue.
- An attempt to produce data such that the output queue's current size plus the produce amount is greater than the queue's capacity.
- An attempt to read, consume or produce data to or from a non-existent node port.
- An attempt to read or consume from a node port which is not connected to a queue.
- An attempt to disconnect a queue from a node port to which it is not connected.
- An attempt to link a graph port to a node port when either port has been previously linked.

4.0 User Interface Description

The user interface for the ESL system is the ESL Editor. This section specifies the capabilities and assumptions for the ESL Editor necessary to produce ESL applications.

4.1 The ESL Editor

The ESL Editor allows the user to create and modify graphs, specify graph attributes, command the storage and retrieval of graphs, and link the primitive nodes to their corresponding executables. The ESL editor is graphical in its operation, using a mouse and pointer to select menu commands and select and manipulate objects on the screen.

As noted in section 2.2 the ESL Editor shall access the Bauhaus system to access and manipulate the Bauhaus schema structure. To access Bauhaus capabilities, the ESL editor shall make use of the ART interfaces. Use of these interfaces shall enable the ESL Editor to read and modify graph information residing in the Bauhaus schemas and retrieve graph data referenced by the Bauhaus schemas. This modification of the schemas is invisible to the user who deals only with the graphs and graph objects appearing on the screen.

4.2 Menu Commands

The following menu commands, though arranged and categorized in a menu-like fashion, are meant to define a list of conceptual capabilities necessary in the editor. As part of a design process prior to any implementation, screen modalities specified by some of the capabilities discussed in this section, would need to be considered. All functionality here is specified in a menu form for consistency and ease of discussion. Command semantics are discussed, particularly where the command would necessitate interaction with the Bauhaus system. An overview of the menu structure is shown in Figure 4.2-1.

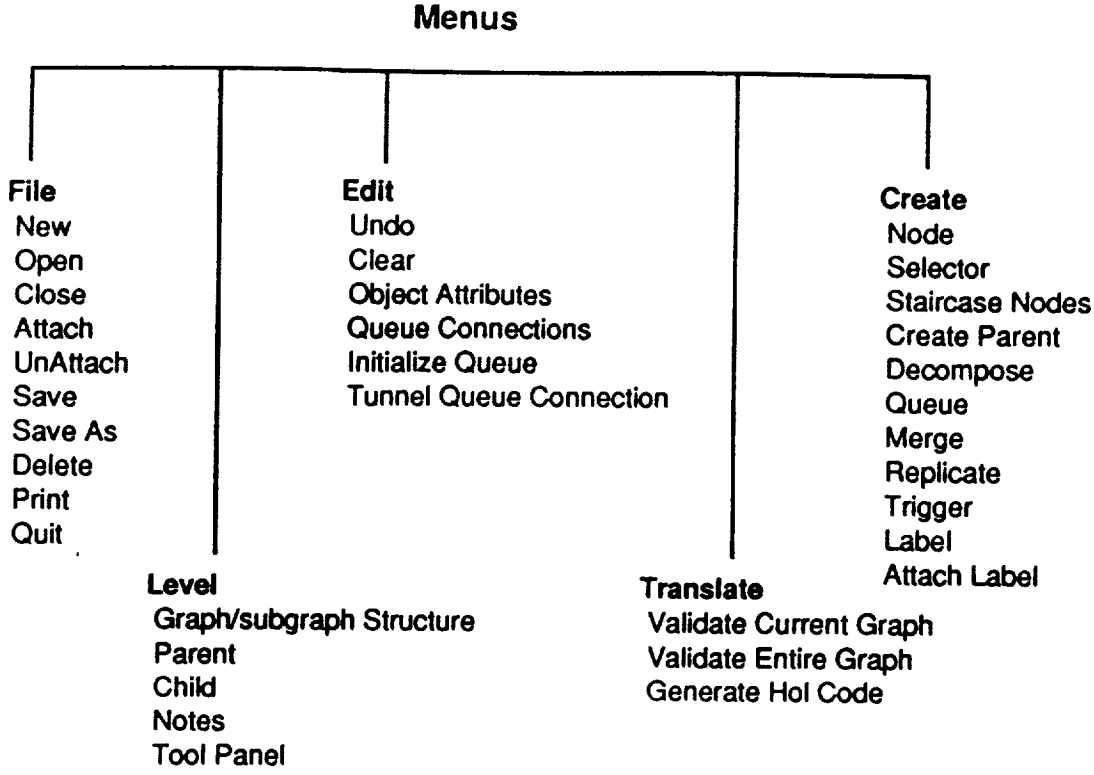


Figure 4.2-1 ESL Menu Overview

4.2.1 File

4.2.1.1 New

This command causes a new (empty) editing window to appear.

It also causes a Bauhaus schema to be instantiated corresponding to this new graph. This schema will initially be assumed to be that of a top level graph.

4.2.1.2 Open

This command causes the selected item to be opened. If the selected item is the name of a graph or subgraph in the Bauhaus taxonomy (Tool Panel window - see 4.2.4.7) , the graph of the selection is shown. If the selected item is the node attached to a subgraph, then the attached subgraph is displayed. If the selected item is the name of a primitive in the taxonomy or the node attached to a primitive, the execution parameters and the ports of that node are shown. If the selected item is a selector node, the selector dialog box appears. If the selected item is an unattached node, a queue, a merge or replicator (box), then this command is dimmed.

4.2.1.3 Close

This command causes the displayed window to close and the files of the graph that was displayed in the window and its associated schema to also close. If there were unsaved changes made to the graph, a dialog box appears asking the user if they wish to save the changes.

4.2.1.4 Attach

This command causes the selected primitive or subgraph selected in the Tool Panel window to be attached to the node selected when the Tool Panel option in the Level menu was chosen. This command is dimmed unless an unattached node on a graph has been selected.

The command also causes the name of the attached subgraph or primitive to be annotated in the Bauhaus schema.

4.2.1.5 UnAttach

This command causes the subgraph or primitive attached to a node to become unattached. This command is dimmed unless an attached node on a graph, has been selected.

This command also causes the name of the (previously) attached subgraph or primitive to be removed from the Bauhaus schema for this node.

4.2.1.6 Save

Choosing this command causes the changes that were made to an application at any level to be saved. If the original application was configuration controlled, then choosing this command will be treated as if the Save As command were chosen and the Save As dialog window will appear, asking for the name to save the application under. A graph that is baselined may not be modified and saved under the same name, but must have a new name and be saved to a nonbaselined file.

The Bauhaus schema is appropriately annotated with the changes that were made. If this graph was referenced by other graphs and this graph was edited under the context of one application, then a separate schema for this graph would be created and the new changes would only be reflected in the newly created schema.

4.2.1.7 Save As

Choosing this command causes the changes that were made to an application at any level to be saved. A Save As dialog box will appear, asking for the name to save the application under; otherwise the semantics are the same as the Save command described above. The user must save the graph in his own directory as he cannot save to a baselined directory.

If the window is saved as a main graph or subgraph, the Bauhaus schema is appropriately annotated. If this new graph is derived from an old one, a new Bauhaus schema is also created. This new schema may reference all appropriate subgraphs and other data, differing only from the original in those aspects of the graph that have been modified.

4.2.1.8 Delete

Choosing this command causes the selected item(s) to be deleted. If nothing is selected then this command is dimmed on the menu.

If the selected items have Bauhaus schema representation then that representation is also deleted. If the selected item is represented by an attribute/attribute pair then that pair is deleted from the appropriate schema. If the deleted item is a node then the schema representing that node is deleted (along with associated table data) and the attribute/attribute value pair on the parent graph denoting the deleted node is also deleted.

4.2.1.9 Print

Choosing this command will cause a dialog window to appear, asking the user if they wish to print only the displayed graph (or table), the entire graph structure, or the entire graph structure with all table information.

4.2.1.10 Quit

Choosing this command causes the ESL Editor to quit. If there have been any changes since the last save was made, a dialog box appears, asking the user if they wish to save the changes.

4.2.2 Edit

4.2.2.1 Undo

Choosing this command causes the effects of the last editing command to be undone. This command applies only to editing commands and not to file manipulation commands such as Save.

4.2.2.2 Clear

Upon the user choosing this command, all ESL components appearing on the screen are deleted. The schema representing the window is not deleted nor is the attribute/attribute value pair referencing the parent schema. However, the attribute/attribute value pairs representing the deleted contents of the screen, are deleted.

4.2.2.3 Object Attributes

This command causes the attributes window of the selected object to appear. If the selected object is a queue then the queue attributes for that particular queue are displayed. If the selected object is a node then the node attributes are displayed, showing the ports the node has, if they are input or output ports, the associated variable names (if they are mapped), how many variable values are read from that port at a time, the offset, and how many are consumed. If the selected object is a selector node then the conditions for selection are shown.

4.2.2.4 Queue Connections

This command causes the display of the Queue Connection Table. This table specifies the sending and receiving ports for each queue (if the inputs have been joined or the outputs have been replicated) identifying the variables. Connections may be modified by the editing of this table. This command is dimmed if no queue is selected.

4.2.2.5 Initialize Queue

Choosing this command causes a dialog box to appear, allowing the user to enter data on the queue. It is the responsibility of the user to ensure that all data is specified to the individual component level and meets all constraints of the type corresponding to this queue. Applications with queues initialized this way will always have the same specified values upon graph initiation until changed by the user. If the graph containing the queue is reused in another application, the queue is still initialized in the same way.

4.2.2.6 Tunnel Queue Connection

When a queue (data flow) is selected and this command is chosen, the graph construct (arrowed line) denoting this queue is hidden. The port-queue tail connection or queue head-port connection may be tunneled separately, causing the queue to appear on one

graph, but not on another. This command is dimmed if no queue is selected. The attribute/attribute values on the schema and the queue listing in the queue connection table, are not affected.

This command is used to control the number of arrowed lines on a graph in an attempt to improve graph readability. One example of the use of this command would be to simplify the appearance of a graph containing a node reading input data from some source and then passing that data to other nodes. Since the input node (the node reading the data) would have many, perhaps hundreds, of connections from its output ports to queues linked to other nodes, this would cause a cluttered and confusing graph if all connections that emanate from the input node, were shown. To control the complexity of the connections on the graph, the outputs from the input node could be tunneled. This would cause the arrows emanating from the input node to be hidden. The arrows denoting the tunnelled queues would only appear on the (sub)graphs containing nodes that receive inputs from those queues. These arrows would appear as if they represented queues that are connected to a graph port , i.e. the arrows have no tails connected to a node on the graph.

4.2.3 Create

4.2.3.1 Node

Choosing this command causes a node to be created on the current graph/subgraph. The created node can then be dragged (using a mouse) to the position desired.

This command causes a subnode attribute to be created on the associated Bauhaus schema. The value of this attribute is the name of the associated subgraph or primitive. This value should reference the schema for the associated subgraph or primitive. The name used for this node is system-assigned initially. The user may change the name of the node by the use of the label command and this will cause a resultant change on the graph schema(s).

4.2.3.2 Selector

Choosing this command causes a selector node to appear on the graph. The selector node may then be dragged (using the mouse) to the desired location.

A selector node is considered a node for the purposes of Bauhaus schema definition.

4.2.3.3 Staircase Nodes

Choosing this option causes a dialog box to appear asking how many nodes need to be created. When this is specified the specified number of nodes will be created and placed on the screen.

Bauhaus schemas are created for each of the created nodes. An attribute for each node is added to the schema for the graph into which the nodes were placed. The name of the attribute value in each case is the name of the created node. The attribute value references the newly created, associated schema for each node.

4.2.3.4 Create Parent

Choosing this command causes a parent graph to be created for the current graph. The created parent graph will have a single node with the input/output of the current graph represented.

A Bauhaus schema for the created parent graph will also be instantiated. This schema will reference the child graph as a subgraph. The schema will have a value of NIL for the attribute value corresponding to the parent attribute.

4.2.3.5 Decompose

Choosing this command while a node on the graph is selected, indicates that the selected node will have a subgraph as the underlying operation. A schema is then instantiated for this node and the schema representing the present graph is then annotated with an attribute/attribute value pair referencing the decomposed node.

4.2.3.6 Queue

This command causes the mouse pointer to become an active attachment device (a change in shape of the pointer also occurs). When the mouse button is clicked while over the output region of a node, a line is anchored on the chosen output region and extends to the mouse pointer wherever the pointer may be moved on the screen. The chosen node is interpreted to be the head end of the queue (and thus the tail of an arrow denoting the queue on the graph). When the mouse button is again clicked while the mouse pointer is over an allowable input region of a node, that node is chosen to be at the tail end of the queue (and thus the arrow will point to this node on the graph).

A queue attribute table representing this queue shall be created. This command also causes a queue attribute to be created on the graph's schema. The name of the attribute value for this attribute shall be the name of the queue. The attribute value shall point to the appropriate queue attribute table.

4.2.3.7 Merge

This command is dimmed unless a queue is selected, i.e. unless an arrow is highlighted on the displayed graph. Choosing this command when a queue is selected, shall cause the nodes on a graph to be highlighted whenever the mouse pointer is placed over them. If the mouse button is clicked while a node is highlighted, that node is selected and an arrow shall be drawn on the graph from the output region of the selected node, to the queue displayed when the command was chosen.

4.2.3.8 Replicate

This command is dimmed unless a queue is selected, i.e. unless an arrow is highlighted on the displayed graph. Choosing this command when a queue is selected, shall cause the nodes on a graph to be highlighted whenever the mouse pointer is placed over them. If the mouse button is clicked while a node is highlighted, that node is selected and an arrow shall be drawn on the graph from the queue displayed when the command was chosen, to the input regions of the selected node.

This command causes the contents of the indicated queue to be checked as part of the scheduling of all the nodes receiving output from this queue.

The amount read and consumed from the queue is logically, separately maintained for all receiving nodes, i.e. information is not dequeued unless all nodes have consumed it.

4.2.3.9 Trigger

The screen semantics of choosing the trigger command are identical to those of the queue command.

4.2.3.10 Label

Choosing this command allows the user to type in a name on the graph. This name is considered as a block of text until attached to a graph object.

4.2.3.11 Attach Label

Choosing this command allows a node, queue, or graph variable to be named. This name replaces the system assigned name on the Bauhaus schema.

4.2.4 Level

4.2.4.1 Graph/subgraph Structure

This command graphically displays the hierarchies of the chosen application.

The Graph/subgraph Structure view is of the whole application or top level graph hierarchy, no matter where in the hierarchy the command is chosen.

4.2.4.2 Parent

Selecting this command will cause the parent graph to be displayed. If no parent node exists, then this command is dimmed.

4.2.4.3 Child

Choosing this command while a node on the displayed graph is selected, causes the subgraph representing the selected node to be displayed. If no node with a subgraph is selected, then this command is dimmed.

4.2.4.4 Queue Attribute Table

This command displays the variable name and capacity of the all queues on the graph or just a selected queue (if only one is selected). This table displays all the queues of a graph, the variables they are labeled with, the ports they are mapped to, and a textual description of each variable.

4.2.4.5 Notes

Choosing this command causes a *Notes* box for the selected graph node to appear. This *Notes* box contains a textual description for the selected node. This command is dimmed until a node has been selected.

If a user wishes to add additional comments to the notes box for a baselined application, then a new notes box is created containing the old textual description plus the changes the user has made, and the attribute value in the node schema is changed to reference this new notes box. The modified application will, of course, be saved in the user's account and not in the baselined directory.

4.2.4.6 Tool Panel

This command causes the Bauhaus Tool Panel windows: Taxonomy, Matches, and Bookmarks to be displayed.

4.2.5 Translate

4.2.5.1 Validate Current Graph

Choosing this command causes various checks of the displayed graph to be made. These checks include type checking when ports are connected through a queue, to ensure that the data structures are compatible and the data will be correctly interpreted by both port processes. If a mismatch occurs, a dialog box stating the mismatch will appear. Another check would ascertain if any queues or ports are not mapped. A dialog box would notify the user of any unmapped queues or ports, however this would not stop the graph from being a valid graph.

4.2.5.2 Validate Entire Graph

This command performs the same checks as the Validate Current Graph, but does so for the entire graph system.

4.2.5.3 Generate HOL Code

This commands, causes the translator to translate the graph schemas representing the current graph to be translated to ESL command statements, in HOL code. A dialog box will appear querying the user for a file name to contain the resultant source code.

5.0 Engineering Scripting Language Command Statements Specification

This section defines and specifies command statements that are to be supported by ESL Shell services. Several Shell services are predefined. They are implemented as either functions (which return a value), or procedures. These command statements may be used in combination with other Ada statements and may be invoked from within a primitive operation.

These commands statements are specified in Ada because of that language's expressive capabilities. This is not meant to mandate the use of Ada, but rather to more precisely specify the characteristics that the ESL Command Statements need to possess.

The specification of the parameter modes for the following interfaces assumes that the run-time structures created by the command statements will be globally visible to the ESL Shell executive processes.

5.1 Command Statement Syntactic and Semantic Rules

These statements are used to build and control graphs. They are organized into the following categories:

- Create graphs and nodes and modify their Node Execution Parameters (NEP).
- Create queues and move data to and from queues.
- Set and remove triggers.
- Requests to the Shell to start/stop executing particular graph systems and provide state information about a node.

5.1.1 Commands to create and modify nodes and NEPs

5.1.1.1 CREATE_GRAPH - Instantiate a Graph

The CREATE_GRAPH function instantiates a graph system structure which is used for nodes which have subgraphs. The function definition of CREATE_GRAPH is:

```
function CREATE_GRAPH (PARENT_NODE: in NODE_ID)
        return NODE_ID;
```

The parameter to the CREATE_GRAPH function is:

PARENT_NODE is the parent node identifier.

Characteristics of the CREATE_GRAPH function are:

- All nodes with subgraphs should have an associated CREATE_GRAPH statement.

- The CREATE_GRAPH statement should appear in the node's underlying operation before any CREATE_NODE statements.

- If no parent graph system is specified for the CREATE_GRAPH operation, then a NON_EXISTING_NODE_ERROR is raised unless the node to be instantiated is the top node in the system. The top node is identified by the PARENT_NODE parameter having the value NIL.

- A node may only have one subgraph associated with it. An attempt to create a graph system for a parent node which already has a subgraph will raise PRE_EXISITING_GRAPH_ERROR.

5.1.1.2 CREATE_NODE - Instantiate a Node

The CREATE_NODE function instantiates a node and specifies the underlying operation. The underlying operation is a procedure identifier. The function definition of CREATE_NODE is:

```
function CREATE_NODE (PARENT_NODE: in  NODE_ID;
        OPERATION_NAME: in STRING) return NODE_ID;
```

The parameters to the CREATE_NODE function are:

PARENT_NODE is the parent node identifier.

OPERATION_NAME is the name of a procedure which specifies the underlying operation structure.

Characteristics of the CREATE_NODE function are:

- The value returned from the CREATE_NODE function is the identifier of the node.

- The the default values for the NEPS are:

READ	1;
OFFSET	0;
CONSUME	1;
THRESHOLD	1;

- If no parent graph system is specified for the CREATE_NODE operation, then a NON_EXISTING_NODE_ERROR is raised unless the node to be instantiated is

the top node in the system. The top node is identified by the parent node having the value NIL.

5.1.1.3 LINK_GRAPH_INPUT_PORT - Connects a Graph Input Port with a Node Input Port

The LINK_GRAPH_INPUT_PORT procedure logically connects a graph input port with node input port. The connectivity is logical since the ports of the graph's parent node are logically the same as certain ports within the graph. The procedure definition of LINK_GRAPH_INPUT_PORT is:

```
procedure LINK_GRAPH_INPUT_PORT (GRAPH_NAME : in NODE_ID;
        GRAPH_PORT: in GRAPH_INPUT_PORT;
        NODE_NAME: in NODE_ID;
        PORT: in INPUT_PORT);
```

The parameters to the LINK_GRAPH_INPUT_PORT procedure are:

GRAPH_NAME is the parent node identifier.

GRAPH_PORT specifies the graph's port.

NODE_NAME is the identifier of a previously created node.

PORT is the port number of the node, NODE_NAME, to be connected.

Characteristics of the LINK_GRAPH_INPUT_PORT procedure are:

- If the graph structure specified by GRAPH_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by GRAPH_PORT does not exist then NON_EXISTING_PORT_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.
- If the port specified by PORT is already connected to another graph port then CONNECTIVITY_ERROR is raised.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.

5.1.1.4 LINK_GRAPH_OUTPUT_PORT - Connects a Graph Output Port with a Node Output Port

The LINK_GRAPH_OUTPUT_PORT procedure logically connects a graph output port with node output port. The connectivity is logical since the ports of the graph's parent node are logically the same as certain ports within the graph. The procedure definition of LINK_GRAPH_OUTPUT_PORT is:

```
procedure LINK_GRAPH_OUTPUT_PORT (GRAPH_NAME : in NODE_ID;
        GRAPH_PORT: in GRAPH_OUTPUT_PORT;
        NODE_NAME: in NODE_ID;
        PORT: in OUTPUT_PORT);
```

The parameters to the LINK_GRAPH_OUTPUT_PORT procedure are:

GRAPH_NAME is the parent node identifier.

GRAPH_PORT specifies the graph's port.

NODE_NAME is the identifier of a previously created node.

PORT is the port number of the node to be connected.

Characteristics of the LINK_GRAPH_OUTPUT_PORT procedure are:

- If the graph structure specified by GRAPH_NAME does not exist then NON_EXISTING_GRAPH_ERROR is raised.
- If the port specified by GRAPH_PORT does not exist then NON_EXISTING_PORT_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.
- If the port specified by PORT is already connected to another graph port then CONNECTIVITY_ERROR is raised.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.

5.1.1.5 GET_READ_NEP - Determines the Value of the READ NEP

The GET_READ_NEP function returns the current value of the READ node execution parameter. The function definition of GET_READ_NEP is:

```
function GET_READ_NEP (NODE_NAME: in NODE_ID;
        PORT: in INPUT_PORT) return READ_AMOUNT;
```

The parameters to the GET_READ_NEP function are:

NODE_NAME is the identifier of the node which owns the NEP.

PORT is the port number of the node for which the NEP is requested.

Characteristics of the GET_READ_NEP function are:

- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.

- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.6 GET_OFFSET_NEP - Determines the Value of the OFFSET NEP

The GET_OFFSET_NEP function returns the current value of the OFFSET node execution parameter. The function definition of GET_OFFSET_NEP is:

```
function GET_OFFSET_NEP (NODE_NAME: in NODE_ID;
        PORT: in INPUT_PORT) return OFFSET_AMOUNT;
```

The parameters to the GET_OFFSET_NEP function are:

NODE_NAME is the identifier of the node which owns the NEP.

PORT is the port number of the node for which the NEP is requested.

Characteristics of the GET_OFFSET_NEP function are:

- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.7 GET_CONSUME_NEP - Determines the Value of the CONSUME NEP

The GET_CONSUME_NEP function returns the current value of the CONSUME node execution parameter. The function definition of GET_CONSUME_NEP is:

```
function GET_CONSUME_NEP (NODE_NAME: in NODE_ID;
        PORT: in INPUT_PORT) return CONSUME_AMOUNT;
```

The parameters to the GET_CONSUME_NEP function are:

NODE_NAME is the identifier of the node which owns the NEP.

PORT is the port number of the node for which the NEP is requested.

Characteristics of the GET_CONSUME_NEP function are:

- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.8 GET_THRESHOLD_NEP - Determines the Value of the THRESHOLD NEP

The GET_THRESHOLD_NEP function returns the current value of the THRESHOLD node execution parameter. The function definition of GET_THRESHOLD_NEP is:

```
function GET_THRESHOLD_NEP (NODE_NAME: in NODE_ID;
        PORT: in INPUT_PORT) return THRESHOLD_AMOUNT;
```

The parameters to the GET_THRESHOLD_NEP function are:

NODE_NAME is the identifier of the node which owns the NEP.

PORT is the port number of the node for which the NEP is requested.

Characteristics of the GET_CONSUME_NEP function are:

- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.9 GET_PRODUCE_NEP - Determines the Value of the PRODUCE NEP

The GET_PRODUCE_NEP function returns the current value of the PRODUCE node execution parameter. The function definition of GET_PRODUCE_NEP is:

```
function GET_PRODUCE_NEP (NODE_NAME: in NODE_ID;
        PORT: in OUTPUT_PORT) return PRODUCE_AMOUNT;
```

The parameters to the GET_PRODUCE_NEP function are:

NODE_NAME is the identifier of the node which owns the NEP.

PORT is the port number of the node for which the NEP is requested.

Characteristics of the GET_PRODUCE_NEP function are:

- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.10 NEW_READ_NEP - Assign a New Value to the READ NEP

The NEW_READ_NEP procedure assigns a new value to the READ node execution parameter which affects how queue data is processed. The procedure definition is:

```
procedure NEW_READ_NEP (NODE_NAME: in NODE_ID;
        PORT: in INPUT_PORT;
        NEW_VALUE: in READ_AMOUNT);
```

The parameters to the NEW_READ_NEP procedure are:

NODE_NAME is the identifier of the node which owns the NEP.

PORT is the port number of the node for which the NEP is requested.

NEW_VALUE is the new value to be assigned to the READ NEP.

Characteristics of the NEW_READ_NEP procedure are:

- The threshold value is automatically changed to the maximum of (READ + OFFSET, CONSUME) if the READ, OFFSET, or CONSUME NEPS are changed and the new value is greater than the current threshold value.
- When a NEP value is specified by means of a command statement then the value is assigned to the node port. If the underlying operation is a subgraph, then the value is inherited by the node port in the subgraph which is linked to the corresponding graph port.
- If the new NEP values would cause the attached queue, if any, to exceed capacity, then CAPACITY_ERROR is raised.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.11 NEW_OFFSET_NEP - Assign a New Value to the OFFSET NEP

The NEW_OFFSET_NEP procedure assigns a new value to the OFFSET node execution parameter which affects how queue data is processed. The procedure definition is:

```
procedure NEW_OFFSET_NEP (NODE_NAME: in NODE_ID;
        PORT: in INPUT_PORT;
        NEW_VALUE: in OFFSET_AMOUNT);
```

The parameters to the NEW_OFFSET_NEP procedure are:

NODE_NAME is the identifier of the node which owns the NEP.

PORT is the port number of the node for which the NEP is requested.

NEW_VALUE is the new value to be assigned to the OFFSET NEP.

Characteristics of the NEW_OFFSET_NEP procedure are:

- The threshold value is automatically changed to the maximum of (READ + OFFSET, CONSUME) if the READ, OFFSET, or CONSUME NEPS are changed and the new value is greater than the current threshold value.
- When a NEP value is specified by means of a command statement then the value is assigned to the node port. If the underlying operation is a subgraph, then the value is inherited by the node port in the subgraph which is linked to the corresponding graph port.
- If the new NEP values would cause the attached queue, if any, to exceed capacity, then CAPACITY_ERROR is raised.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.12 NEW_CONSUME_NEP - Assign a New Value to the CONSUME NEP

The NEW_CONSUME_NEP procedure assigns a new value to the CONSUME node execution parameter which affects how queue data is processed. The procedure definition is:

```
procedure NEW_CONSUME_NEP (NODE_NAME: in NODE_ID;
        PORT: in INPUT_PORT;
        NEW_VALUE: in CONSUME_AMOUNT);
```

The parameters to the NEW_CONSUME_NEP procedure are:

NODE_NAME is the identifier of the node which owns the NEP.

PORT is the port number of the node for which the NEP is requested.

NEW_VALUE is the new value to be assigned to the CONSUME NEP.

Characteristics of the NEW_CONSUME_NEP procedure are:

- The threshold value is automatically changed to the maximum of (READ + OFFSET, CONSUME) if the READ, OFFSET, or CONSUME NEPS are changed and the new value is greater than the current threshold value.
- When a NEP value is specified by means of a command statement then the value is assigned to the node port. If the underlying operation is a subgraph, then the

value is inherited by the node port in the subgraph which is linked to the corresponding graph port.

- If the new NEP values would cause the attached queue, if any, to exceed capacity, then CAPACITY_ERROR is raised.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.13 NEW_THRESHOLD_NEP - Assign a New Value to the THRESHOLD NEP

The NEW_THRESHOLD_NEP procedure assigns a new value to the THRESHOLD node execution parameter which affects how queue data is processed. The procedure definition is:

```
procedure NEW_THRESHOLD_NEP (NODE_NAME: in NODE_ID;
        PORT: in INPUT_PORT;
        NEW_VALUE: in THRESHOLD_AMOUNT);
```

The parameters to the NEW_THRESHOLD_NEP procedure are:

NODE_NAME is the identifier of the node which owns the NEP.

PORT is the port number of the node for which the NEP is requested.

NEW_VALUE is the new value to be assigned to the THRESHOLD NEP.

Characteristics of the NEW_THRESHOLD_NEP procedure are:

- When a NEP value is specified by means of a command statement then the value is assigned to the node port. If the underlying operation is a subgraph, then the value is inherited by the node port in the subgraph which is linked to the corresponding graph port.
- The exception THRESHOLD_ERROR is raised if the threshold value is specified to be less than the maximum of (READ + OFFSET, CONSUME).
- If the new NEP values would cause the attached queue, if any, to exceed capacity, then CAPACITY_ERROR is raised.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.14 NEW_PRODUCE_NEP - Assign a New Value to the PRODUCE NEP

The NEW_PRODUCE_NEP procedure assigns a new value to the PRODUCE node execution parameter which affects how queue data is processed. The procedure definition is:

```
procedure NEW_PRODUCE_NEP (NODE_NAME: in NODE_ID;
        PORT: in OUTPUT_PORT;
        NEW_VALUE: in PRODUCE_AMOUNT);
```

The parameters to the NEW_PRODUCE_NEP procedure are:

NODE_NAME is the identifier of the node which owns the NEP.

PORT is the port number of the node for which the NEP is requested.

NEW_VALUE is the new value to be assigned to the PRODUCE NEP.

Characteristics of the NEW_PRODUCE_NEP procedure are:

- When a NEP value is specified by means of a command statement then the value is assigned to the node port. If the underlying operation is a subgraph, then the value is inherited by the node port in the subgraph which is linked to the corresponding graph port.
- If the new NEP values would cause the attached queue, if any, to exceed capacity, then CAPACITY_ERROR is raised.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.15 SET_NEPS - Assigns New Values to All Node Execution Parameters

The SET_NEPS procedure assigns new values to the node execution parameters which affect how queue data is processed. The procedure definition is:

```
procedure SET_NEPS (NODE_NAME: NODE_ID;
        PORT: INPUT_PORT;
        NEW_READIN: READ_AMOUNT;
        NEW_OFFSET: OFFSET_AMOUNT;
        NEW_CONSUME: CONSUME_AMOUNT;
        NEW_THRESHOLD: THRESHOLD_AMOUNT);
```

The parameters to the SET_NEPS procedure are:

NODE_NAME is the identifier of the node which owns the NEPS

PORT is the port number of the node for which the NEP is requested.

NEW_READIN, NEW_OFFSET, NEW_CONSUME, NEW_THRESHOLD are the new NEP values to be assigned to the node.

Characteristics of the SET_NEPS procedure are:

- The threshold value is automatically changed to the maximum of (READ + OFFSET, CONSUME) if the new value is greater than the current threshold value.
- When a NEP value is specified by means of a command statement then the value is assigned to the node port. If the underlying operation is a subgraph, then the value is inherited by the node port in the subgraph which is linked to the corresponding graph port.
- If the new NEP values would cause the attached queue, if any, to exceed capacity, then CAPACITY_ERROR is raised.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.1.16 GET_PRIORITY - Determines Priority Level of Node

The GET_PRIORITY function returns the priority level value. The function definition is:

```
function GET_PRIORITY (NODE_NAME: in NODE_ID)
        return PRIORITY_LEVEL;
```

The parameter to the GET_PRIORITY function is:

NODE_NAME is the identifier of the node which owns the NEPS.

Characteristics of the GET_PRIORITY function are:

- The values of PRIORITY_LEVEL are implementation defined.
- The value 0 is an undefined, default priority level which may be associated with nodes which have subgraphs as their underlying operation.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.

5.1.1.17 NEW_PRIORITY - Assigns New Priority Level to Node

The NEW_PRIORITY procedure set a new priority level value. The function definition is:

```
procedure NEW_PRIORITY (NODE_NAME: in NODE_ID;
        PRIORITY: in PRIORITY_LEVEL);
```

The parameters to the NEW_PRIORITY procedure are:

NODE_NAME is the identifier of the node which owns the NEPS.

PRIORITY is the new priority level.

Characteristics of the NEW_PRIORITY procedure are:

- The values of PRIORITY_LEVEL are implementation defined.
- The number of priority levels is implementation dependent.
- The new priority level of a node with subgraphs as the underlying operation is inherited by all nodes within the subgraph. The inheritance process continues down the hierarchy until all nodes with underlying primitive operations have been assigned the new priority.
- The use of this procedure allows nodes within the same graphs system to have different priority levels.
- This procedure may be used to statically set the priority of a node at instantiation time or dynamically set the priority of a node at execution time.
- For nodes already scheduled to execute or currently executing, the binding of the new priority level will take effect after the node has executed.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.

5.1.2 Commands to start and stop graph execution and query the status of a node.

5.1.2.1 NODE_READY - Determines Ready State of Node

The NODE_READY function allows the ready state of a node to be evaluated. The function definition of NODE_READY is:

```
function NODE_READY (NODE_NAME: in NODE_ID) return BOOLEAN;
```

The parameter to the NODE_READY function is:

NODE_NAME is the identifier of the node for which the ready state will be determined.

Characteristics of the NODE_READY function are:

- NODE_READY determines if the node satisfies the scheduling criteria, not if the node has been scheduled for execution. A node may be ready, but not scheduled, if NODE_NAME is the identifier of the currently executing node or the execution of the node has not been started or has been stopped.
- NODE_READY returns a value of TRUE if the node is ready for execution, FALSE if the node is not ready for execution.
- If this function is invoked with NODE_NAME being the identifier of the currently executing node and returns the value TRUE, then the execution of the underlying primitive operation may be repeated without requiring the node to be rescheduled by the Shell.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.

5.1.2.2 START_NODE - Starts the Execution of a Graph System

The START_NODE procedure submits an instantiated graph system to the Shell for execution. The procedure definition of START_NODE is:

```
procedure START_NODE (PARENT_NODE: in NODE_ID;
        STATUS_INDICATOR: out BOOLEAN);
```

The parameters to the START_NODE procedure are:

PARENT_NODE is the identifier of the graph system.

STATUS_INDICATOR returns whether the procedure was successful.

Characteristics of the START_NODE procedure are:

- PARENT_NODE must have been instantiated.
- If the node specified by PARENT_NODE does not exist then NON_EXISTING_NODE_ERROR is raised.

5.1.2.3 STOP_NODE - Stops the Execution of a Graph System

The STOP_NODE procedure causes the Shell to stop executing a graph system. The procedure definition of STOP_NODE is:

```
procedure STOP_NODE (PARENT_NODE: in NODE_ID
        STATUS_INDICATOR: out BOOLEAN);
```

The parameters to the STOP_NODE procedure are:

PARENT_NODE is the identifier of the graph system.

STATUS_INDICATOR returns whether the procedure was successful.

Characteristics of the STOP_NODE procedure are:

- PARENT_NODE must be a valid identifier which represents a currently executing graph system.
- No queue data shall be lost as a result of this operation.
- Queues connected to the parent node shall remain connected.
- If the node specified by PARENT_NODE does not exist then NON_EXISTING_NODE_ERROR is raised.

5.1.3 Commands to control the setting and consuming of triggers

5.1.3.1 PRODUCE_TRIGGER - Produces Triggers onto an Attached Output Queue

The PRODUCE_TRIGGER procedure allows triggers to be written to the tail of a queue attached to an output port. The procedure definition of PRODUCE_TRIGGER is:

```
procedure PRODUCE_TRIGGER (CURRENT_NODE: in NODE_ID;
        OUT_PORT: in OUTPUT_PORT;
        AMOUNT: in CAPACITY_AMOUNT);
```

The parameters to the PRODUCE_TRIGGER procedure are:

CURRENT_NODE is the node identifier of the currently executing node.

OUT_PORT is the output port number of the node in which the data is produced.

AMOUNT is the number of triggers to be placed on the tail of the queue connected to OUT_PORT.

Characteristics of the PRODUCE_TRIGGER procedure are:

- The amount of data to be produced is specified by the PRODUCE NEP associated with the node's output port.
- If the output port is not connected to a queue, then the data is discarded.
- If the node specified by CURRENT_NODE does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by OUT_PORT does not exist then NON_EXISTING_PORT_ERROR is raised.
- A CAPACITY_ERROR if the added triggers would cause the queue attached to OUT_PORT to go over capacity.

5.1.3.2 ENQUEUE_TRIGGER - Produces Triggers onto an Unattached Queue

The ENQUEUE_TRIGGER procedure allows triggers to be written to the tail of a queue. The queue does not have to be attached to the node. The procedure definition of ENQUEUE_TRIGGER is:

```
procedure ENQUEUE_TRIGGER (QUEUE_NAME: in QUEUE_ID;
        AMOUNT: in CAPACITY_AMOUNT);
```

The parameters to the ENQUEUE_TRIGGER procedure are:

QUEUE_NAME is the identifier of a previously created queue which is to have data added to its tail.

AMOUNT is the number of data items to be extracted from the buffer variable and placed onto the queue.

Characteristics of the ENQUEUE_TRIGGER procedure are:

- A CAPACITY_ERROR if the added triggers would cause the queue to go over capacity.
- If the queue specified by QUEUE_NAME does not exist then NON_EXISTING_QUEUE_ERROR is raised.

5.1.3.3 CONSUME_TRIGGER - Consumes Triggers from an Attached Input Queue

The CONSUME_TRIGGER procedure allows triggers to be removed from the head of queue attached to an input port. The procedure definition of CONSUME_TRIGGER is:

```
procedure CONSUME_TRIGGER (CURRENT_NODE: in NODE_ID;
        IN_PORT: in INPUT_PORT;
        AMOUNT: in CAPACITY_AMOUNT);
```

The parameters to the CONSUME_TRIGGER procedure are:

CURRENT_NODE is the node identifier of the currently executing node.

IN_PORT is the input port number of the node in which the data is read.

AMOUNT is the number of triggers to be removed from the head of the queue connected to IN_PORT.

Characteristics of the CONSUME_TRIGGER procedure are:

- The amount of data to be consumed is specified by the CONSUME NEP associated with the node's input port.

- The consume operation is destructive to the data on the queue.
- The input port must have a queue attached or a CONNECTIVITY_ERROR is raised.
- If the node specified by CURRENT_NODE does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by IN_PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.3.4 FLUSH_TRIGGER - Removes Triggers from an Unattached Queue

The FLUSH_TRIGGER procedure allows triggers to be removed from an unattached queue. The procedure definition of FLUSH_TRIGGER is:

```
procedure FLUSH_TRIGGER (QUEUE_NAME: in QUEUE_ID;
        AMOUNT: in CAPACITY_AMOUNT);
```

The parameters to the FLUSH_TRIGGER procedure are:

QUEUE_NAME is the identifier of a previously created queue which is to have triggers removed.

AMOUNT is the number of triggers to be removed from the head of the queue.

Characteristics of the FLUSH_TRIGGER procedure are:

- The predefined amount ALL causes all data to be removed from the queue.
- Should the queue contain less data items than the amount, an INSUFFICIENT_DATA_ERROR results.
- If the queue specified by QUEUE_NAME does not exist then NON_EXISTING_QUEUE_ERROR is raised.

5.1.3.5 INIT_TRIGGER_Q - Initializes an Unattached Queue with Triggers

The INIT_TRIGGER_Q procedure removes all triggers from a queue and allows triggers to be written to the tail of a queue. The queue does not have to be attached to a node. The procedure definition of INIT_TRIGGER_Q is:

```
procedure INIT_TRIGGER_Q (QUEUE_NAME: in QUEUE_ID;
        AMOUNT: in MAX_QUEUE_SIZE);
```

The parameters to the INIT_TRIGGER_Q procedure are:

QUEUE_NAME is the identifier of a previously created queue which is to be initialized.

AMOUNT is the number of data items to be extracted from the buffer variable and placed onto the queue.

Characteristics of the INIT_TRIGGER_Q procedure are:

- A CAPACITY_ERROR is raised if the added data would cause the queue to go over capacity.
- Prior to placing new triggers on the queue, the queue is flushed of all existing triggers.
- If the queue specified by QUEUE_NAME does not exist then NON_EXISTING_QUEUE_ERROR is raised.

5.1.4 Commands to create and move data to and from queues

5.1.4.1 Q_SIZE - Determines Queue Size

The Q_SIZE function returns the current size of a queue. The size of a queue refers to the number of data elements on the queue. The function definition of Q_SIZE is:

```
function Q_SIZE (QUEUE_NAME: in QUEUE_ID) : return INTEGER;
```

The parameter to the Q_SIZE function is:

QUEUE_NAME is the queue identifier of the queue for which its size is requested.

A characteristic of the Q_SIZE function is:

- If the queue specified by QUEUE_NAME does not exist then NON_EXISTING_QUEUE_ERROR is raised.

5.1.4.2 CREATE_Q - Create a Queue

The CREATE_Q function creates a queue structure. The maximum length of the queue is specified as well as the type of data elements which may be transported through the queue. The function definition of CREATE_Q is:

```
function CREATE_Q (CAPACITY: in CAPACITY_AMOUNT)
        return QUEUE_ID;
```

The parameters to the CREATE_Q function are:

CAPACITY is an integer amount of the maximum number of data elements which the queue can have at any one time.

A characteristics of the CREATE_Q function is:

- The value returned from the CREATE_Q function is the identifier of the queue.

5.1.4.3 CONNECT_INPUT_Q - Connect a Queue to a Node

The CONNECT_INPUT_Q procedure links the head of a queue structure to the input port of a node. The procedure definition of CONNECT_INPUT_Q is:

```
procedure CONNECT_INPUT_Q (QUEUE_NAME : in QUEUE_ID;
        NODE_NAME : in NODE_ID;
        PORT : in INPUT_PORT);
```

The parameters to the CONNECT_INPUT_Q procedure are:

QUEUE_NAME is the identifier of a previously created queue.

NODE_NAME is the sink node identifier of a previously created node.

PORT is the port number of the node in which the queue head is to be connected.

Characteristics of the CONNECT_INPUT_Q procedure are:

- A queue that is connected to a node will be considered in the node scheduling criteria. If a node is scheduled for execution or is executing at the time a queue is connected to it, then the node will continue to execute regardless of the amount of data on the queue.
- A queue may be attached to several input ports (for several nodes). Consuming data from a shared queue by the execution of node may affect the execution eligibility of other nodes. The Shell must check the execution eligibility of other nodes receiving input from the shared queue anytime data is consumed from the queue.
- If the queue specified by QUEUE_NAME does not exist then NON_EXISTING_QUEUE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.
- If the queue's capacity is insufficient to support the NEPs of the port of NODE_NAME specified by PORT, then CAPACITY_ERROR is raised..
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.

5.1.4.4 CONNECT_OUTPUT_Q - Connect a Queue to a Node

The CONNECT_OUTPUT_Q procedure links the tail of a queue structure to the output port of a node. The procedure definition of CONNECT_OUTPUT_Q is:

```
procedure CONNECT_OUTPUT_Q (QUEUE_NAME : in QUEUE_ID;
        NODE_NAME : in NODE_ID;
        PORT : in OUTPUT_PORT);
```

The parameters to the CONNECT_OUTPUT_Q procedure are:

QUEUE_NAME is the identifier of a previously created queue.

NODE_NAME is the source node identifier of a previously created node.

PORT is the port number of the node in which the queue head or tail is to be connected.

Characteristics of the CONNECT_OUTPUT_Q procedure are:

- A queue that is connected to a node will be considered in the node scheduling criteria. If a node is scheduled for execution or is executing at the time a queue is connected to it, then the node will continue to execute regardless of the amount of data on the queue.
- Multiple queues may be attached to the same output port. Queues attached to output ports are distinct, yet they will receive the same data during a PRODUCE_DATA operations.
- If the queue specified by QUEUE_NAME does not exist then NON_EXISTING_QUEUE_ERROR is raised.
- If the port specified by PORT does not exist then NON_EXISTING_PORT_ERROR is raised.
- If the queue's capacity is insufficient to support the NEPs of the port specified by PORT, then CAPACITY_ERROR is raised.
- If the node specified by NODE_NAME does not exist then NON_EXISTING_NODE_ERROR is raised.

5.1.4.5 PRODUCE_DATA - Produces Data onto an Attached Output Queue

The PRODUCE_DATA procedure allows data to be written to the tail of a queue attached to an output port. The procedure definition of PRODUCE_DATA is:

```
procedure PRODUCE_DATA (CURRENT_NODE in NODE_ID;
        DATA_ARRAY: in OUTPUT_PORT;
        OUT_ARRAY: in BUFFER);
```

The parameters to the PRODUCE_DATA procedure are:

CURRENT_NODE is the node identifier of the currently executing node.

OUT_PORT is the output port number of the node that is producing the data.

DATA_ARRAY is the array variable into which the data will be written by the node that is producing the data.

Characteristics of the PRODUCE_DATA procedure are:

- The amount of data to be produced is specified by the PRODUCE NEP associated with the node's output port.
- The data elements to be placed on the queue must be of the same data type used in the CREATE_Q function. They are added to the queue after all elements already on the queue. Data taken from the buffer will be placed onto the queue tail, element by element, starting with the buffer element index 1 being placed onto the tail of the queue, and continuing, with increasing buffer array index, until the specified number of elements has been placed into the queue. The remainder of the queue will not be disturbed.
- If an output port is not connected to a queue, then the data is discarded.
- An output port, which has multiple queues attached, will result in the same data being produced onto each queue.
- If the node specified by CURRENT_NODE does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by OUT_PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.4.6 ENQUEUE_DATA - Produces Data onto an Unattached Queue

The ENQUEUE_DATA procedure allows data to be written to the tail of a queue. The queue does not have to be attached to the node. The procedure definition of ENQUEUE_DATA is:

```
procedure ENQUEUE_DATA (QUEUE_NAME: in QUEUE_ID;
        DATA_ARRAY: in BUFFER;
        AMOUNT: in CAPACITY_AMOUNT);
```

The parameters to the ENQUEUE_DATA procedure are:

QUEUE_NAME is the identifier of a previously created queue which is to have data added to its tail.

DATA_ARRAY is an initialized array variable which contains the data to be placed onto the tail of the queue.

AMOUNT is the number of data items to be extracted from the buffer variable and placed onto the queue.

Characteristics of the ENQUEUE_DATA procedure are:

- The data elements to be placed on the queue must be of the same data type used in the CREATE_Q function. They are added to the queue after all elements already on the queue. Data taken from the buffer will be placed onto the queue tail, element by element, starting with the buffer element index 1 being placed onto the tail of the queue, and continuing, with increasing buffer array index, until the specified number of elements has been placed into the queue. the remainder of the queue will not be disturbed.

- If the added data would cause the queue to go over capacity then CAPACITY_ERROR is raised.

- If the queue specified by QUEUE_NAME does not exist then NON_EXISTING_QUEUE_ERROR is raised.

5.1.4.7 INIT_DATA_Q - Initializes an Unattached Queue with Data

The INIT_DATA_Q procedure allows all data to be removed from a queue and data to be written to the tail of a queue. The queue does not have to be attached to a node. The procedure definition of INIT_DATA_Q is:

```
procedure INIT_DATA_Q (QUEUE_NAME: in QUEUE_ID;
        VAR_BUFFER: in BUFFER;
        AMOUNT: in CAPACITY_AMOUNT);
```

The parameters to the INIT Q procedure are:

QUEUE_NAME is the identifier of a previously created queue which is to be initialized.

VAR_BUFFER is a properly initialized variable which represents the data to be placed onto the queue.

AMOUNT is the number of data items to be extracted from the buffer variable and placed onto the queue.

Characteristics of the INIT_DATA_Q procedure are:

- The data elements to be placed on the queue must be of the same data type used in the CREATE_Q function. They are added to the queue after all elements already on the queue. Data taken from the buffer will be placed onto the queue tail, element by element, starting with the buffer element index 1 being placed onto the tail of the queue, and continuing, with the increasing buffer array index, until the specified number of elements has been placed into the queue. The remainder of the queue will not be disturbed.

- Prior to placing new data objects on the queue, the queue is flushed of all existing data.
- If the added data would cause the queue to go over capacity then CAPACITY_ERROR is raised.
- If the queue specified by QUEUE_NAME does not exist then NON_EXISTING_QUEUE_ERROR is raised.

5.1.4.8 READ_DATA - Read Data from an Attached Input Queue

The READ_DATA procedure causes data to be written from the head of the attached queue into a variable. The READ_DATA operation may be indexed into the queue by the offset amount. The procedure definition of READ_DATA is:

```
procedure READ_DATA (CURRENT_NODE: in NODE_ID;
        IN_PORT: in INPUT_PORT;
        VAR_BUFFER: out BUFFER);
```

The parameters to the READ_DATA procedure are:

CURRENT_NODE is the node identifier of the currently executing node.

IN_PORT is the input port number of the node in which the data is read.

VAR_BUFFER is the variable in which the read data will be stored.

Characteristics of the READF_DATA procedure are:

- The buffer variable must be large enough to store the data being read.
- The amount of data to be read is specified by the READ NEP associated with the nod's input port. The starting position of the read is indexed by the offset amount which is specified by the OFFSET NEP associated with the node's input port.
- The read operation is non-destructive to the data on the queue.
- The input port must have a queue attached or the exception CONNECTIVITY_ERROR will be raised.
- If the node specified by CURRENT_NODE does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by IN_PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.4.9 MOVE_DATA - Move Data from an Input Queue to an Output Queue

The MOVE_DATA procedure allows data to be moved directly from an input queue to an output queue of the currently executing node. The procedure definition of MOVE_DATA is:

```
procedure MOVE_DATA (CURRENT_NODE: in NODE_ID;
        IN_PORT: in INPUT_PORT;
        OUT_PORT: in OUPUT_PORT;
        AMOUNT: in CAPACITY_AMOUNT);
```

The parameters to MOVE_DATA are:

CURRENT_NODE is the identifier of the currently executing node which has the input and output queues attached to it.

IN_PORT is the port number associated with the input queue.

OUT_PORT is the port number associated with the output queue.

AMOUNT is the number of data items to be moved from the input queue to the output queue.

Characteristics of the MOVE_DATA procedure are:

- The amount of data is removed from the input queue and added to the output queue.
- The order of the data is unchanged.
- CAPACITY_ERROR is raised if the current size of the input queue is less than the amount or if the current size of the output queue plus the amount would cause the queue to exceed capacity.
- If the node specified by CURRENT_NODE does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by IN_PORT does not exist then NON_EXISTING_PORT_ERROR is raised.
- If the port specified by OUT_PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.4.10 CONSUME_DATA - Consumes Data from an Attached Input Queue

The CONSUME_DATA procedure allows data to be removed from the head of queue attached to an input port. The procedure definition of CONSUME_DATA is:

```
procedure CONSUME_DATA (CURRENT_NODE: in NODE_ID;
        IN_PORT: in INPUT_PORT);
```

The parameters to the CONSUME_DATA procedure are:

CURRENT_NODE is the node identifier of the currently executing node.

IN_PORT is the input port number of the node in which the data is read.

Characteristics of the CONSUME_DATA procedure are:

- The amount of data to be consumed is specified by the CONSUME NEP associated with the node's input port.
- The consume operation is destructive to the data on the queue.
- The input port must have a queue attached or CONNECTIVITY_ERROR will be raised.
- If the node specified by CURRENT_NODE does not exist then NON_EXISTING_NODE_ERROR is raised.
- If the port specified by IN_PORT does not exist then NON_EXISTING_PORT_ERROR is raised.

5.1.4.11 DEQUEUE_DATA - Reads Data from an Unattached Queue

The DEQUEUE_DATA procedure allows data to be read with an offset from an unattached queue. The procedure definition of DEQUEUE is:

```
procedure DEQUEUE_DATA (QUEUE_NAME: in QUEUE_ID;
        VAR_BUFFER: out BUFFER;
        AMOUNT: in CAPACITY_AMOUNT;
        OFFSET: in OFFSET_AMOUNT);
```

The parameters to the DEQUEUE_DATA procedure are:

QUEUE_NAME is the identifier of a previously created queue which is to have data added to its tail.

VAR_BUFFER is the variable in which the data to be stored.

AMOUNT is the number of data items to be read from the queue's head.

OFFSET is the index amount into the queue's head.

Characteristics of the DEQUEUE_DATA procedure are:

- INSUFFICIENT_DATA_ERROR is raised if the amount + offset is greater than the current queue size.
- If the queue specified by QUEUE_NAME does not exist then NON_EXISTING_QUEUE_ERROR is raised.

5.1.4.12 FLUSH_DATA - Removes Data from an Unattached Queue

The FLUSH_DATA procedure allows data to be removed from an unattached queue. The procedure definition of FLUSH_DATA is:

```
procedure FLUSH_DATA (QUEUE_NAME: in QUEUE_ID;
        AMOUNT: in CAPACITY_AMOUNT);
```

The parameters to the FLUSH_DATA procedure are:

QUEUE_NAME is the identifier of a previously created queue which is to have data removed.

AMOUNT is the number of data items to be removed from the queue's head.

Characteristics of the FLUSH_DATA procedure are:

- The predefined amount ALL causes all data to be removed from the queue.
- Should the queue contain less data items than the amount, INSUFFICIENT_DATA_ERROR will be raised.
- If the queue specified by QUEUE_NAME does not exist then NON_EXISTING_QUEUE_ERROR is raised.

5.2 ESL Command Specification

```
package ESL_COMMANDS is

   type QUEUE_ID is private;

   MAX_QUEUE_SIZE: constant INTEGER := implementation defined;

   STATUS_INDICATOR: BOOLEAN;


   type NODE_ID is private;

   type PRIORITY_LEVEL is private;

   type NEP is private;

   type READ_AMOUNT is range  1 .. MAX_QUEUE_SIZE;

   type OFFSET_AMOUNT is range  1 .. MAX_QUEUE_SIZE;
```

```
   type CONSUME_AMOUNT is range  1 .. MAX_QUEUE_SIZE;
   type CAPACITY_AMOUNT is range  1 .. MAX_QUEUE_SIZE;
   type THRESHOLD_AMOUNT is range  1 .. MAX_QUEUE_SIZE;
   type PRODUCE_AMOUNT is range  1 .. MAX_QUEUE_SIZE;

   type PORT_ID is private;
   subtype INPUT_PORT is PORT_ID;
   subtype OUTPUT_PORT is PORT_ID;
   type GRAPH_PORT_ID is private;
   subtype INPUT_GRAPH_PORT is GRAPH_PORT_ID;
   subtype OUTPUT_GRAPH_PORT is GRAPH_PORT_ID;
-------------------------------------------------------
--    QUEUE and DATA FLOW COMMANDS    --
-------------------------------------------------------
   function Q_SIZE (QUEUE_NAME: in QUEUE_ID) : return INTEGER;
generic
   type QDATA is private;
package QUEUE_COMMANDS is
   type BUFFER is array(1 .. MAX_QUEUE_SIZE) of QDATA;
   function CREATE_Q (CAPACITY: in CAPACITY_AMOUNT)
            return QUEUE_ID;
   procedure CONNECT_INPUT_Q (QUEUE_NAME : in QUEUE_ID;
            NODE_NAME : in NODE_ID;
            PORT : in INPUT_PORT);
   procedure CONNECT_OUTPUT_Q (QUEUE_NAME : in QUEUE_ID;
            NODE_NAME : in NODE_ID;
            PORT : in OUTPUT_PORT);
```

```
   procedure PRODUCE_DATA (CURRENT_NODE in NODE_ID;
            OUT_PORT: in PORT_ID;
            BUFFER: in QDATA);

   procedure ENQUEUE_DATA (QUEUE_NAME: in QUEUE_ID;
            BUFFER: in QDATA;
            AMOUNT: in INTEGER);

   procedure INIT_DATA_Q (QUEUE_NAME: in QUEUE_ID;
            VAR_BUFFER: in BUFFER;
            AMOUNT: in CAPACITY_AMOUNT);

   procedure READ_DATA (CURRENT_NODE: in NODE_ID;
            IN_PORT: in PORT_ID;
            VAR_BUFFER: out BUFFER);

   procedure MOVE_DATA (CURRENT_NODE: in NODE_ID;
            IN_PORT: in PORT_ID;
            OUT_PORT: in PORT_ID;
            AMOUNT: in CAPACITY_AMOUNT);

   procedure CONSUME_DATA (CURRENT_NODE: in NODE_ID;
            IN_PORT: in PORT_ID);

   procedure DEQUEUE_DATA (QUEUE_NAME: in QUEUE_ID;
            VAR_BUFFER: out BUFFER;
            AMOUNT: in CAPACITY_AMOUNT;
            OFFSET: in OFFSET_AMOUNT);

   procedure FLUSH_DATA (QUEUE_NAME: in QUEUE_ID;
            AMOUNT: in CAPACITY_AMOUNT);

end QUEUE_COMMANDS;

-------------------------------------------------------

--   TRIGGER COMMANDS   --

-------------------------------------------------------

   procedure PRODUCE_TRIGGER (CURRENT_NODE: in NODE_ID;
            OUT_PORT: in OUTPUT_PORT;
            AMOUNT: in CAPACITY_AMOUNT);

   procedure ENQUEUE_TRIGGER (QUEUE_NAME: in QUEUE_ID;
            AMOUNT: in INTEGER);

   procedure CONSUME_TRIGGER (CURRENT_NODE: in NODE_ID;
            IN_PORT: in INPUT_PORT;
            AMOUNT: in CAPACITY_AMOUNT);
```

```
   procedure FLUSH_TRIGGER (QUEUE_NAME: in QUEUE_ID;
            AMOUNT: in CAPACITY_AMOUNT);

   procedure INIT_TRIGGER_Q (QUEUE_NAME: in QUEUE_ID;
            AMOUNT: in CAPACITY_AMOUNT);

-------------------------------------------------------

--    NODES COMMANDS     --

-------------------------------------------------------

   function CREATE_GRAPH (PARENT_NODE: in NODE_ID)
            return NODE_ID;

   function CREATE_NODE (PARENT_NODE: in  NODE_ID;
            OPERATION_NAME : in STRING) return NODE_ID;

   procedure LINK_PRIMITIVE (NODE_NAME : in NODE_ID;
            PRIMITIVE_OPERATION : in STRING);

   procedure LINK_GRAPH_PORT (GRAPH_NAME : in NODE_ID;
            GRAPH_PORT: in GRAPH_PORT_ID;
            NODE_NAME: in NODE_ID;
            PORT: in PORT_ID);

   function GET_READ_NEP (NODE_NAME: in NODE_ID;
            PORT: in PORT_ID;) return READ_AMOUNT;

   function GET_OFFSET_NEP (NODE_NAME: in NODE_ID;
            PORT: in PORT_ID;) return OFFSET_AMOUNT;

   function GET_CONSUME_NEP (NODE_NAME: in NODE_ID;
            PORT: in PORT_ID;) return CONSUME_AMOUNT;

   function GET_THRESHOLD_NEP (NODE_NAME: in NODE_ID;
            PORT: in PORT_ID;) return THRESHOLD_AMOUNT;

   function GET_PRODUCE_NEP (NODE_NAME: in NODE_ID;
            PORT: in OUTPUT_PORT) return PRODUCE_AMOUNT;

   procedure NEW_READ_NEP (NODE_NAME: in NODE_ID;
            PORT: in PORT_ID;
            NEW_VALUE: in READ_AMOUNT);

   procedure NEW_OFFSET_NEP (NODE_NAME: in NODE_ID;
            PORT: in PORT_ID;
            NEW_VALUE: in OFFSET_AMOUNT);
```

```
   procedure NEW_CONSUME_NEP (NODE_NAME: in NODE_ID;
            PORT: in PORT_ID;
            NEW_VALUE: in CONSUME_AMOUNT);

   procedure NEW_THRESHOLD_NEP (NODE_NAME: in NODE_ID;
            PORT: in PORT_ID;
            NEW_VALUE: in THRESHOLD_AMOUNT);

   procedure NEW_PRODUCE_NEP (NODE_NAME: in NODE_ID;
            PORT: in OUTPUT_PORT;
            NEW_VALUE: in PRODUCE_AMOUNT);

   procedure SET_NEPS (NODE_NAME: NODE_ID;
            PORT: PORT_ID;
            NEW_READIN: READ_AMOUNT;
            NEW_OFFSET: OFFSET_AMOUNT;
            NEW_CONSUME: CONSUME_AMOUNT;
            NEW_THRESHOLD: THRESHOLD_AMOUNT);

   function GET_PRIORITY (NODE_NAME: in NODE_ID)
            return PRIORITY_LEVEL;

   procedure NEW_PRIORITY (NODE_NAME: in NODE_ID;
            PRIORITY: in PRIORITY_LEVEL);

-------------------------------------------------------

--   EXECUTION COMMANDS   --

-------------------------------------------------------

   function NODE_READY (NODE_NAME: in NODE_ID) return BOOLEAN;

   procedure START_NODE (PARENT_NODE: in NODE_ID;
            STATUS_INDICATOR: out BOOLEAN);

   procedure STOP_NODE (PARENT_NODE: in NODE_ID
            STATUS_INDICATOR: out BOOLEAN);

-------------------------------------------------------

--   EXCEPTION LIST   --

-------------------------------------------------------

   PRE_EXISTING_GRAPH_ERROR: exception;

   NON_EXISTING_NODE_ERROR: exception;

   NON_EXISTING_GRAPH_ERROR: exception;
```

```
   NON_EXISTING_PORT_ERROR: exception;
   NON_EXISTING_QUEUE_ERROR: exception;
   CONNECTIVITY_ERROR: exception;
   CAPACITY_ERROR: exception;
   THRESHOLD_ERROR: exception;
   INSUFFICIENT_DATA_ERROR: exception;
private
--  implementation defined
end ESL_COMMANDS;
```

6.0 Recommendations for Further Research & Development

This section presents alternatives and directions for further investigation. Some of these suggestions are intended to answer the questions necessary to construct an operational ESL system. Other suggestions address directions for further research efforts.

It cannot be too strongly stated that the primary goal of all ensuing efforts be directed toward the construction of an operational prototype of the ESL system and the development of a realistic application. The prototype application should be developed using existing application models/algorithms that have been redesigned to take advantage of the ESL's structured concepts. It is felt that these measures are necessary to gain support for ESL in the Mission Operations Directorate (MOD) user community.

A study of commercial products that may satisfy the ESL requirements should be made. In the area of control systems analysis and simulation, several products are available[3]. While the commercial products may not satisfy all requirements, the potential for cost savings are significant. The user can augment the functions of many tools, for example, adding new primitives and library elements for code generation. Possible advantages of this approach are: 1) reduced cost ; 2) faster deployment of useful systems for operations; 3) larger installed base of users; and 4) user groups. The commercial product study should assess the feasibilty of the various development approaches and their ability to satisfy the requirements of the ESL with respect to cost, reliability, maintainability, and product support.

6.1 Domain Analyses

• What are the application areas that interest MOD the most? Do these application areas indicate a short term need for ESL? Or would the need be a long term one?

• What is the proper granularity size for ESL primitive development? This investigation would need to address both the intended uses by the anticipated user(s) as well as the need assessment raised by the preceding question.

• Is MOD the only customer? A simulation processer such as ESL could be used to verify expert systems and neural networks as replacement for algorithmic procedures.

6.2 Environment and Capability Specification

• Is simulation at the workstation during an editing session desired? If so, the Shell executive would need to be designed to be callable from the ESL Editor and to dynamically allocate the necessary graph storage and map the addresses of data to the primitives that need that data.

• Is dynamic graph manipulation desired? If it is intended that graph connections and instances be modified during execution, the set of command statements must be extended to

[3]for example, Matrix/X, among others.

allow the real-time disconnect of ports to queues and the linking of previously compiled procedures to newly created nodes.

• Are extensive graph debugging capabilities desired? The capability to capture run-time information such as remaining queue capacities and overflows, perform single stepping and other debugger functionality, would need to be designed into the executive.

6.3 The Translator

The translation process discussed in Section 2.0 is a brief description of a design/implementation dependent process that must be specified before moving into those phases. To be able to specify the translation process several decisions about the implementation must first be made. A few of these design alternatives are listed below:

• What is the language of implementation? The features of the chosen language will greatly influence the ultimate design. If the language is C for instance, then the Shell executive will likely use procedure passing (as a parameter) to pass in the file names of primitives. If the language is Ada, what will be the language construct used to represent primitives: procedures or tasking?

• What tools will be available for use and what are their capabilties? If the Bauhaus text manipulation tool is used (and is capable), then templates written in the desired language could be used and the names of different graph objects would be substituted in. For instance, a template of source code that checks unspecified queues and calls unspecified procedures could be used as part of the executive. The translation process would substitute the names of the queues and primitives indicated in the graph schema, into the templates. Note that this method would allow the building of the graph implementation simultaneously with the graph schema, another design decision.

Appendix A - Scenarios

Contained in this appendix are scenarios that were used to develop requirements for the ESL Editor and its integration with the Inference Bauhaus system. The intent by including these scenarios in this report, is to provide some insight to the reader as to how some of the basic operations using the ESL Editor, were intended to be performed. This is not meant to be a limitation on the future designers or implementors, but rather an informal dialog on the concepts in the writers/reviewers minds at this time. There were four scenarios used for initial conceptualization.

A.1 Access the knowledge base from the ESL Editor.

While in the Editor, the user chooses the Tool Panel menu selection to display the Bauhaus windows. The user then chooses to look at Application (Top) Level graphs, Subgraphs, or Primitives. The selection of one of these will cause the appropriate subclasses/instances to appear in the other window. The semantics of the operation of the windows is exactly the same as that of the Bauhaus. When the user wishes a file to be displayed, they choose Open from the File menu while the appropriate file is selected or use other means provided by the Bauhaus enviroment to open applications. This causes the selected graph to be displayed in the Editor window.

A.2 Editing Graphs

A.2.1 Modifying an existing application by loading a subgraph or primitive.

The user selects an application graph from the opening tool panel screen and starts the ESL editor (the exact method of starting the ESL Editor is undefined since it lies outside of the ESL definition). The top level graph appears in the ESL window. The user descends through the graph structure until they come to a node that they wish to modify and then select that node. If the node had been previously attached and the user wished to another subgraph (or primitive) he must first unattach that node by choosing Unattach from the File menu. To attach that node to a subgraph or primitive the user first chooses Tool Panel from the Level menu. This causes the tool panel windows to appear over the ESL window. The user chooses Subgraphs (or Primitives) on the top Tool Panel window and proceeds down through the subgraph/primitive hierarchy until finding the appropriate subgraph/primitive. To examine the selected subgraph the user Opens it. When the user is sure that he has the appropriate subgraph/primitive, he selects Attach from the File menu and the subgraph/primitive is loaded and the Tool Panel window is closed.

A.2.2 Saving a subgraph

Regardless of which of the following methods is chosen, the Bauhaus schema associated with the graph would be appropriately modified with the changes that were made to the graph. Multiple applications may use a baselined graph or subgraph. However, a graph or subgraph that is baselined may not be modified and saved under the same name; it must be saved to a nonbaselined file. If a subgraph was edited under the context of a particular application, and this subgraph is referenced by other applications, then a separate schema

for this subgraph would be created and the new changes would only be reflected on the newly created schema. This would insure that no other application was affected by the changes made to the subgraph. If a change to a subgraph is needed and it is desired for this change to be reflected on all referencing applications, then the baselined graph must be modified by an engineer with the proper access.

A.2.2.1 Using the Save As command

With the graph to be saved displayed on the screen, the user chooses the Save As menu command. Choosing this command causes the changes that were made to an application at any level, to be saved. A Save As dialog box will appear, asking for the name to save the application under.. A graph that is baselined may not be modified and saved under the same name. Instead it must be saved under a new name in the user's local directory.

If the window is saved as a main graph or subgraph, the Bauhaus schema is appropriately modified. If this new graph is derived from an old one, a new Bauhaus schema is also created. This new schema may reference all appropriate subgraphs and associated data, differing only from the original in those aspects of the graph that have been modified.

A.2.2.2 Using the Save command

With the graph to be saved displayed on the screen, the user chooses the Save menu command. Choosing this command causes the changes that were made to an application, at any level, to be saved. If the original application was configuration controlled, then choosing this command will be treated as if the Save As command were chosen and the Save As dialog window will appear, asking for the name to save the application under.

A.2.3 Matching subgraphs/primitives

All issues about matching have not been resolved.

The Match command would work on attribute/attribute value pairs. One thought on the subject would allow the items to be matched on to be selectable. In that case Match should have its own menu command with a pop up window. Can the Bauhaus search mechanism handle this?

A.3 Providing data values.

A.3.1 Using primitives to get data

The options discussed in this section will work if the graph has one input primitive that reads in all data and distributes it to all other primitives or if each primitive will read in its own data.

A.3.1.1 Predefined file names hard coded into the primitive.

In this case the name of a file is hardcoded into an initialization primitive (here a primitive is assumed, however, a subgraph for reading in the data could just as easily be used). This name is the file name of the input data file or a file of pointers to the input data. The user muds make sure that the initialization primitive is attached to a node in the application graph and that the node output ports are connected to the proper queues for that application.

When the application was started, the primitive would read the input data out of the file and place all the data upon the queues and then trigger the next node to be executed.

A.3.1.2 The file name is passed to the primitive.

This option would necessitate another primitive reading in the name of the input data file and passing it to the initialization node. The file name may be provided as a user query made by a query node. This query node could be a generic node instantiated many times to ask for different inputs.

A.3.2 Using the Initialize Queue menu option

This option allows the user to select a queue on the graph and deposit an initial value(s) into that queue. This would be accomplished by the selection of the Initialize Queue menu command while the desired queue was selected. Choosing this command would cause a dialog box to appear, allowing the user to enter data onto the queue. It is the responsibility of the user to ensure that all data is specified to the (scalar) component level and meets all constraints of the type corresponding to this queue. Applications with queues initialized this way will always have the same specified values upon graph initiation until changed by the user. If the graph containing the queue is reused in another application, the queue is still initialized in the same way.

A.4 Reading a user input form using a primitive

This option would use a primitive to read a form generated for user inputs. The issues and merits for using this approach are the same as the **Using primitives to get data** option discussed above. This primitive would be specially designed to read user input forms. Taking this approach would put the actual form generation process outside of the ESL Editor The primitive would have read the form. The issue of the identification of the file where the form resides is the same as for any input file.